# PASCAL

## (Teil 1)

Dr. Claus Kofer
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

### 0. Einführung

PASCAL wurde um 1970 von N. Wirth an der ETH Zürich als Lehrsprache für die Programmierausbildung entwickelt. Heute ist sie in der Welt eine der am meisten angewendeten Programmiersprachen. Dafür gibt es eine Reihe von Gründen: PASCAL ist universell anwendbar, hat eine klare Struktur, ist einfach zu erlernen, ermöglicht die effektive Compilierung und steht auf fast allen Rechnern in den wichtigsten Betriebssystemen zur Verfügung.
PASCAL-Programme sind in wesentlich geringerem Maße betriebssystem- und rechnerabhängig als Programme anderer Programmiersprachen. Sie gestatten damit die Entwicklung übertragbarer und langlebiger Softwarelösungen.
Seit 1983 gibt es das TURBO-PASCAL-System. Es wurde von der BORLAND Corp., USA, für den Einsatz auf Arbeitsplatz- und Personalcomputern entwickelt.
Auch in der DDR ist es für alle Büro- und Personalcomputer, die unter einem CP/M-kompatiblen Betriebssystem laufen, verfügbar.
Im Vergleich mit den bis 1983 bekannten PASCAL-Compilern besticht TURBO-PASCAL durch seine außerordentlich hohe Übersetzerleistung von ca. 1000 Zeilen Quellprogramm/Minute. Dies und ein integrierter Quelltexteditor führen zu einem sehr kurzen Programmentwicklungszyklus: Editieren, Compilieren, Testen. Die übersetzten PASCAL-Programme sind sehr kompakt, so daß ihre maximale Größe bei ca. 2000 Quellzeilen liegt. Der Sprachumfang von TURBO-PASCAL entspricht dem von Jensen/Wirth in „PASCAL: User Manual an Report" dargelegten Standard-PASCAL. Es gibt wenig Einschränkungen, aber viele nützliche Erweiterungen wie

- Datentyp STRING
- Direktzugriffsfiles
- Konstanten der strukturierten Datentypen
- Überlagerungsmechanismus für Prozeduren
- Auslösen von Betriebssystemrufen
- Aufruf von Maschinenkodeunterprogrammen
- Einfügen von Maschinenkodepassagen.

Die Grundlage für diesen Beitrag bildet der Themenplan einer Weiterbildungsveranstaltung zur Anwendung der Programmiersprache PASCAL, die der Autor seit mehreren Jahren für EDV-Anwender durchführt. Die einzelnen Elemente der Sprache werden aufeinander aufbauend vorgestellt und ihre Anwendung an Beispielen demonstriert.
Wo es notwendig ist, auf Details einzugehen, wird die Lösung von TURBO-PASCAL geschildert.
Darüber hinaus werden Einschränkungen und Besonderheiten dieses Systems bei der Behandlung der einzelnen PASCAL-Elemente besprochen.
Alle angegebenen Programmbeispiele laufen und sind mit TURBO-PASCAL auf PC 1715 getestet worden.
Den Abschluß bildet ein Kapitel, in dem Erweiterungen und Spezifika von TURBO-PASCAL vorgestellt werden.

### 1. Darstellung der Sprache

Ein Kurs zur Vermittlung einer Programmiersprache muß zwei Aufgaben lösen: Erstens sind darzustellen die vorhandenen Sprachelemente und die an sie geknüpften Inhalte, d. h. die Semantik, und zweitens in welcher Form die Sprachelemente korrekt niederzuschreiben und zu kombinieren sind, die Syntax. Während zur Darstellung der Syntax formale Beschreibungsverfahren angewendet werden, ist es immer noch üblich, die Semantik verbal darzulegen. Eine besonders prägnante Form der Syntaxbeschreibung sind Syntaxdiagramme. Sie wurden erstmals im Zusammenhang mit PASCAL in breitem Umfang genutzt. Auch in diesem Beitrag werden sie angewendet.
Das Bild 1.1 zeigt vier Grundformen von Syntaxdiagrammen. Bei der *Sequenz* (Bild 1.1a) wird a durch eine Folge von b und c gebildet. Während bei einer *Alternative* (Bild 1.1b) a entweder durch b oder durch c gebildet wird. Natürlich können Sequenz und Alternative um Elemente d, e, f, ... angereichert werden, falls es erforderlich ist. In einer *Liste* (Bilder 1.1c und 1.1d) wird a durch eine zunächst beliebig lange Folge von b gebildet. Während die Liste nach Bild 1.1c jedoch mindestens einmal b enthält, kann die nach Bild 1.1d gebildete Liste auch leer sein.
Die Anwendung der Grundformen wird im Bild 1.2 am Beispiel des Syntaxdiagramms für die Repeat-Anweisung gezeigt. Sie enthält im Inneren eine nicht leere Liste von Anweisungen, die voneinander durch das Semikolon getrennt sind. Die Sequenz von REPEAT, der Anweisungsliste, UNTIL und „ausdruck" bildet schließlich die komplette Repeat-Anweisung. Syntaktisch korrekte Formen entstehen durch eine Traversierung des Syntaxdiagrammes in Richtung der Pfeile. So ist die Folge „REPEAT anweisung; anweisung UNTIL ausdruck" korrekt. Als Programmstück wird sie in einer ebenfalls erlaubten anderen Anordnung

```
REPEAT
anweisung;
anweisung;
UNTIL ausdruck
```

sofort deutlich sichtbar.
Das Syntaxdiagramm für die Repeat-Anweisung enthält die klein geschriebenen deutschen Wörter „anweisung" und „ausdruck", die groß geschriebenen englischen Wörter REPEAT und UNTIL sowie das Semikolon. Während die Wörter REPEAT, UNTIL und das Semikolon unmittelbar im fertigen PASCAL-Programm erscheinen dürfen, verweisen „anweisung" und „ausdruck" auf weitere Syntaxdiagramme.
Die Namensgebung für Syntaxdiagramme ist beliebig. Günstig ist es aber, solche Namen zu wählen, die Assoziationen zur Sematik des entsprechenden Syntaxdiagramms herstellen. So wird bei der Repeat-Anweisung deutlich, daß in ihrem Inneren weitere Anweisungen eingelagert sein können.
In diesem Kurs wird durchgängig eine Darstellung gewählt, in der Syntaxdiagramme mit klein geschriebenen Wörtern der deutschen Sprache bezeichnet werden.

*Bild 1.1 Grundformen von Syntaxdiagrammen*
*a) Sequenz*
*b) Alternative*
*c) Liste mit mindestens einem Element*
*d) Liste, die auch leer sein kann*

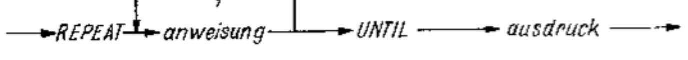


*Bild 1.2 Syntaxdiagramm der Repeat-Anweisung*

### 2. Grundelemente von PASCAL

#### 2.1. Überblick

Die Grundelemente der Programmiersprache PASCAL können in die Klassen

- **Bezeichner**
- **Zahlen**
- **Zeichenketten**
- **Operatoren und spezielle Symbole**

eingeteilt werden.

### 2.2. *Bezeichner*

Bezeichner dienen zur Benennung von Objekten eines PASCAL-Programms. Die zu benennenden Objekte können Datentypen, Daten und Programmkodes sein.

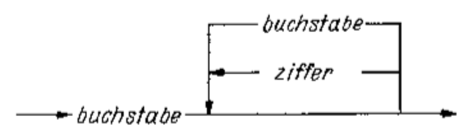



*Bild 2.1 Syntaxdiagramm „bezeichner"*

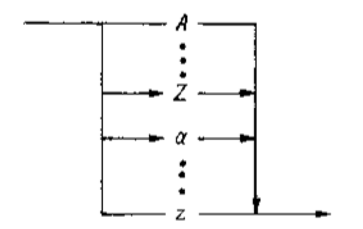



*Bild 2.2 Syntaxdiagramm „buchstabe"*




*Bild 2.3 Syntaxdiagramm „ziffer"*

Die Bildung von Bezeichnern zeigt das Syntaxdiagramm im Bild 2.1. Zur Substitution von „buchstabe" und „ziffer" sind die Syntaxdiagramme nach Bild 2.2. und Bild 2.3. heranzuziehen. Korrekte Bezeichner sind:

Max, x1, a23x, AnfWert, Count

Die Anzahl der signifikanten Stellen eines Bezeichners ist i. allg. begrenzt. Bei TURBO-PASCAL darf ein Bezeichner nicht länger als die Zeilenlänge von 127 Zeichen sein. Weiterhin werden innerhalb eines Bezeichners Groß- und Kleinbuchstaben nicht unterschieden. So sind z. B. die Bezeichner XY und xy gleich.
In PASCAL haben eine Reihe von Bezeichnern eine schon feststehende Bedeutung. Sie sind reserviert und dürfen vom Anwender zur Benennung seiner Objekte nicht benutzt werden:

| | | |
|---|---|---|
| ABSOLUTE(*) | FUNCTION | RECORD |
| AND | GOTO | REPEAT |
| ARRAY | IF | SET |
| BEGIN | IN | SHL(*) |
| CASE | INLINE(*) | SHR(*) |
| CONST | LABEL | STRING(*) |
| DIV | MOD | THEN |
| DO | NIL | TO |
| DOWNTO | NOT | TYPE |
| ELSE | OF | UNTIL |
| END | OR | VAR |
| EXTERNAL(*) | OVERLAY(*) | WHILE |
| FILE | PACKED | WITH |
| FOR | PROCEDURE | XOR(*) |
| FORWARD | PROGRAM | |

Die durch (*) gekennzeichneten Bezeichner sind zusätzlich in TURBO-PASCAL reserviert.

### 2.3. *Zahlen*

Zahlen sind namenlose Datenobjekte mit einem festen numerischen Wert. In PASCAL werden ganze und reelle Zahlen unterschieden. Die Bildung erfolgt nach den Syntaxdiagrammen der Bilder 2.4 und 2.5. Korrekte Beispiele für ganze bzw. reelle Zahlen sind:

12, 10, 1000, 123
bzw.
1.0, 1.5E–03, 121E+14

Zur Unterstützung der maschinennahen Programmierung können ganze Zahlen in hexadezimaler Schreibweise angegeben werden. Es sind dann zusätzlich die Ziffern A, B ..., F bzw. a, b, ..., f erlaubt. Gekennzeichnet wird die hexadezimale Schreibweise durch ein vorangestelltes Dollarzeichen (bzw. das Zeichen ○)
○ 100, ○ 1FF, ○ FFFF

### 2.4. *Zeichenketten*

Eine Zeichenkette repräsentiert ein Datenobjekt, dessen Wert eine Folge von Zeichen des ASCII-Kodes ist. Innerhalb eines PASCAL-Programms kann sie ähnlich wie eine Zahl verwendet werden.
Die Bildung von korrekten Zeichenketten zeigt das Syntaxdiagramm im Bild 2.6. Dabei darf eine Zeichenkette nicht über das Zeilenende hinweg fortgesetzt werden. Korrekte Zeichenketten sind:

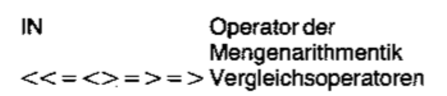
'Testprogramm Version 1.0', 'x:='

Als Spezialfall ist auch die leere Zeichenkette '' zugelassen

### 2.5. *Operatoren und spezielle Symbole*

Operatoren geben die mit den Datenobjekten durchzuführenden Verknüpfungen an. In PASCAL sind folgende Operatoren definiert:

| | |
|---|---|
| := | Zuweisungsoperator |
| + – * / | Arithmetische Grundoperationen |
| DIV | Division ohne Rest |
| MOD | Divisionsrest |
| NOT AND OR | Grundoperationen der Booleschen Arithmetik |
| IN | Operator der Mengenarithmentik |
| <<=<>=>=> | Vergleichsoperatoren |



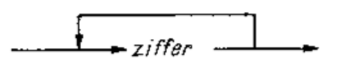



*Bild 2.4 Syntaxdiagramm „ganze zahl"*

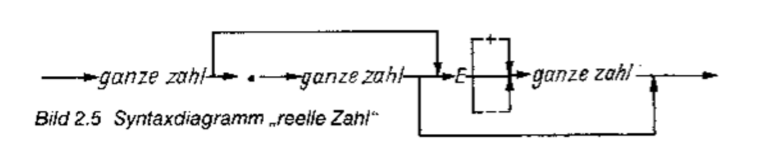



*Bild 2.5 Syntaxdiagramm „reelle Zahl"*

Die Grundoperatoren verknüpfen in der Regel zwei Datenobjekte gleichen Typs und liefern wieder ein Resultat von diesem Typ. Vergleichsoperatoren ergeben unabhängig vom Typ der Operanden stets einen Booleschen Wert.
In TURBO-PASCAL gibt es eine Reihe von weiteren Operatoren. Sie werden später zusammen mit den Datentypen, auf die sie anwendbar sind, besprochen.
Weiterhin gibt es in PASCAL eine Reihe von speziellen Symbolen, die aus syntaktischen Gründen notwendig sind:

| | |
|---|---|
| () | Klammerung der Parameterliste von Prozeduren bzw. Funktionen |
| [] | Klammerung der Indizes von Arrays |
| {} oder (**) | Klammerung von Kommentaren |

Kommentare haben normalerweise keine Auswirkung auf ein Programm. In PASCAL ist es jedoch möglich, mit sogenannten Pseudokommentaren Compileroptions anzugeben, die die Arbeitsweise des Compilers beeinflussen. Eine Option veranlaßt den Compiler z. B., den Zugriff auf Arrays so zu übersetzen, daß die Überschreitung der Indexgrenzen zum definierten Programmabbruch führt. Es gibt weitere Compileroptions, die im Zusammenhang mit den entsprechenden Sprachelementen behandelt werden.
Compileroptions beginnen mit einem Dollarzeichen. Dann folgen ein Buchstabe, der die entsprechende Option charakterisiert und entweder ein Plus- oder ein Minuszeichen, das anzeigt, ob sie aktiviert werden soll oder nicht.

### 2.6. *Format von PASCAL-Programmen*

An das Format von PASCAL-Programmen werden nur zwei Forderungen gestellt:

– Die Grundelemente der Sprache sind durch mindestens ein Leerzeichen oder einen Kommentar zu trennen. Falls die Eindeutigkeit erhalten bleibt, kann das Leerzeichen auch entfallen.
– Bezeichner dürfen nicht durch Zeilenende zerteilt werden.

Insbesondere zur Verbesserung der Lesbarkeit können zwischen den Grundelementen beliebig viele Leerzeichen oder Leerzeilen eingefügt werden. Die PASCAL-Passage

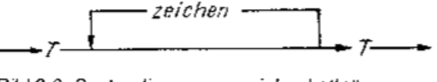



*Bild 2.6 Syntaxdiagramm „zeichenkette"*

a + 1.0 bleibt beim Weglassen der Leerzeichen eindeutig. Das ist jedoch bei REPEAT A; B UNTIL C nicht so.

### 2.7. *Programmbeispiel*

Die Anordnung der einzelnen Grundelemente zu einem kompletten PASCAL-Programm zeigt der nachfolgende Programmtext. Das Programm tabelliert die Funktion
$y = f(x) = x^2$
für ganzzahlige x im Intervall 0 . . max.

```
PROGRAM Tabelle
                    (INPUT, OUTPUT);
VAR x, y, max.:  INTEGER;
BEGIN
  READLN (INPUT, max);
  FOR x: = 0 TO max DO BEGIN
    y: = x*x;
    WRITELN (OUTPUT,'x: = ', x,
    'y: = ', y)
  END
END
```

Die Aufteilung des Programmtextes auf die einzelnen Zeilen kann auch in anderer Weise erfolgen. Ebenso ist das Einrücken von Zeilen nicht zwingend. Die gezeigte Form wurde ausschließlich mit dem Ziel gewählt, die inhaltliche Zusammengehörigkeit der Programmteile auch optisch zum Ausdruck zu bringen.
Zur Konstruktion des Programms wurden folgende Grundelemente verwendet:

Bezeichner:
PROGRAMM INPUT Tabelle VAR × y max INTEGER BEGIN READLN FOR TO DO WRITELN END
Zahlen:
0
Zeichenketten:
'x: = ''y: = '
Operatoren und spezielle Symbole:
; . : ( ) : = *.

## 3. Datentypen

### 3.1. *Überblick*

Die Aufgabe von Datentypen ist es, Wertebereiche für Datenobjekte anzugeben. PASCAL stellt eine Reihe von *Standarddatentypen* zur Verfügung. Zusätzlich können vom Programmierer in gewissen Grenzen *problemangepaßte Datentypen* deklariert werden.

Die Standardtypen sind
- ganze Zahlen (INTEGER)
- reelle Zahlen (REAL)
- Boolesche Werte (BOOLEAN)
- Zeichenkodes des Rechners (CHAR)

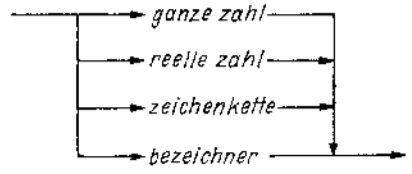



*Bild 2.7 Syntaxdiagramm „konstante"*

Die Wertebereiche der Standardtypen werden durch die Datenformate der zugrundeliegenden Gerätetechnik bestimmt. Hier gibt es Unterschiede zwischen den PASCAL-Systemen verschiedener Rechner.
Der Programmierer kann Aufzählungs- und Teilbereichstypen deklarieren. Für *Aufzählungstypen* sind die Elemente des Wertebereiches explizit zu nennen. Bei *Teilbereichstypen* wird eine Eingrenzung auf einen Teil eines umfassenderen Wertebereiches vorgenommen.
Als weiterer Datentyp werden sogenannte *Mengen* bereitgestellt.
Mit allen bisher genannten Datentypen lassen sich die Strukturen *File, Array* und *Rekord* bilden. Während Files und Arrays aus Datenobjekten gleichen Typs bestehen, können zu Rekords Datenobjekte unterschiedlichen Typs zusammengefaßt werden. Mit dem Datentyp File und darauf anwendbaren Prozeduren und Funktionen wird in PASCAL das gesamte Problem der Ein- und Ausgabe gelöst.
*Pointertypen* gestatten in Abhängigkeit vom Programmablauf das dynamische Anlegen von Datenobjekten aller bisher genannten Typen. Der Wert eines Pointertyps selbst ist eine Hauptspeicheradresse.
Die bisher genannten Datentypen gehören zum Standard-PASCAL. Darüber hinaus gibt es in einzelnen Systemen nützliche Erweiterungen. In TURBO-PASCAL sind das
- ganze Zahlen im Byteformat (BYTE)
- Zeichenketten (STRING).

### 3.2. *Einordnung in die Sprache*

Datentypen werden durch das Syntaxdiagramm „typ" deklariert. Es wird im Bild 3.1 gezeigt. Die Standard-, Aufzählungs- und Teilbereichstypen sind zum Syntaxdiagramm „einfacher typ" zusammengefaßt worden und werden im Bild 3.2 gezeigt. Das hat ausschließlich einen praktischen Grund: An verschiedenen Stellen der Sprache PASCAL sind nicht alle Typen von Datenobjekten zugelassen, sondern nur Standard-, Aufzählungs- und Teilbereichstypen.

## 4. Einfache Typen

### 4.1. *Standardtypen*

#### 4.1.1. *Datentypen INTEGER und REAL*

Der Wertebereich des Datentyps INTEGER umfaßt positive und negative ganze Zahlen. Der Bereich selbst hängt von der internen Verarbeitungsbreite des Rechners ab. Bei den Büro- und Arbeitsplatzrechnern ist sie 16 Bit. Damit wird ein Bereich –32 768 . . . . 32 767 überstrichen.
Die interne Darstellung des Standardtyps REAL erfolgt in Mantisse und Exponent. Die verwendete Anzahl von Bytes ist bei den einzelnen PASCAL-Systemen unterschiedlich. Bei TURBO-PASCAL sind es sechs: eins für den Exponenten und fünf für die Mantisse. Da die Mantisse selbst normalisiert abgespeichert wird, entspricht das einer Darstellung in 40 Bit. Dem entsprechen elf signifikante Stellen in der Dezimalschreibweise.
Für alle Datentypen ist die Ausführung der arithmetischen Grundoperation +, –, * und /

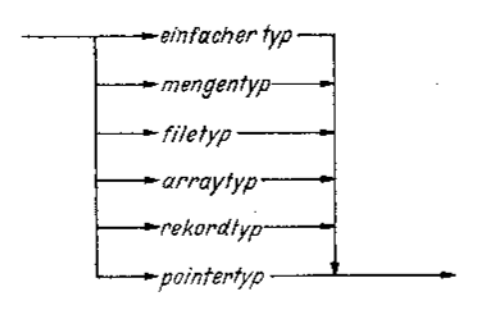



*Bild 3.1 Syntaxdiagramm „typ"*

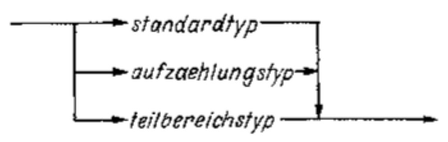



*Bild 3.2 Syntaxdiagramm „einfacher typ"*

und der Vergleiche <, <=, <>, =, > = und > möglich. Für den Datentyp INTEGER gibt es zusätzlich die ganzzahlige Division DIV und den Divisionsrest MOD.
Bei Büro- und Arbeitsplatzrechnern werden die arithmetischen Grundoperationen für den Datentyp REAL softwaremäßig realisiert. Sie sind deshalb deutlich langsamer.
Das Verhalten von PASCAL-Programmen bei *Überschreitung des erlaubten Wertebereiches* ist vom Datentyp abhängig. Bei INTEGER äußert sich eine Überschreitung wegen der internen Darstellung negativer Zahlen im Zweierkomplement als Vorzeichenwechsel und wird bei der Programmabarbeitung nicht „entdeckt".
Bei *Unterschreitung* der betragsmäßig kleinsten REAL-Zahl wird das entsprechende Datenelement auf den Wert 0.0 gesetzt und die Abarbeitung des Programms fortgesetzt. Überschreitet der Betrag den größten darstellbaren Wert, bricht das Programm mit einer entsprechenden Fehlerausschrift ab.
In TURBO-PASCAL werden bestimmte Operationen des Maschinenbefehlssystems auf das Niveau von PASCAL gehoben. Sie stehen als Operatoren SHL, SHR, OR und XOR zum Schieben bzw. für Bitoperationen zur Verfügung und können auf Datenelemente vom Typ INTEGER angewendet werden.

#### 4.2.1. *Datentyp BOOLEAN*

Den Wertebereich des Datentyps BOOLEAN bilden die Wahrheitswerte FALSE und TRUE. Es sind die Operationen NOT, OR und AND sowie alle Vergleiche möglich, wobei FALSE < TRUE ist.
Intern wird ein Byte zur Darstellung benötigt, wobei 00 den Wert FALSE und 01 TRUE repräsentiert.

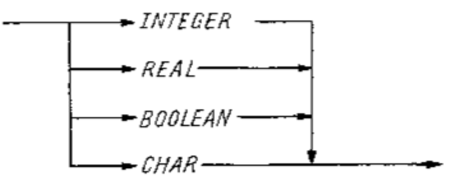



*Bild 4.1 Syntaxdiagramm „standardtyp"*

### 4.1.3. *Datentyp CHAR*

Der Wertebereich des Datentyps CHAR ist der Zeichenkode des jeweiligen Rechners. Beim PC 1715 ist das der ASCII-Kode. Die interne Darstellung erfordert ein Byte. Mit Datenobjekten zum Typ CHAR können keine arithmetischen Verknüpfungen durchgeführt werden. Zuweisung und Vergleichsoperationen sind erlaubt. Vergleichsoperationen orientieren sich an der internen Kodierung der einzelnen Zeichen.
Konstanten vom Typ CHAR werden in Apostrophe eingeschlossen, z. B. 'A', 'B', '+', '*', '>', '0', '1' usw.

### 4.2. *Aufzählungstypen*

Aufzählungstypen werden durch explizite Angabe ihres Wertebereiches deklariert. Bild 4.2 zeigt die Syntax. Die durch Komma getrennten Bezeichner bilden den Wertebereich. Ein Beispiel zeigt dies:

(ROT, BLAU, GRUEN).

Ein Datenobjekt von diesem Aufzählungstyp kann nur einen der drei möglichen Werte ROT oder BLAU oder GRUEN annehmen.

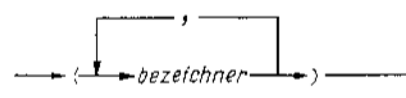



*Bild 4.2 Syntaxdiagramm „aufzaehlungstyp"*

Weitere Beispiele für Aufzählungstypen sind

(Mo, Di, Mi, Don, Fr, Sa, So)
(kleiner, gleich, grösser)
(Lesen, Schreiben, Positionieren).

Durch die Reihenfolge bei der Deklaration wird gleichzeitig eine Ordnung zwischen den einzelnen Werten eines Aufzählungstyps eingeführt. Für die genannten Beispiele gilt:

ROT < BLAU < GRUEN
Mo < Di < Mi < Don < Fr < Sa < So
usw.

Die Ausführung arithmetischer Operationen ist mit Aufzählungstypen nicht möglich. Es sind nur Zuweisung und Vergleichsoperationen erlaubt.
Werden mehrere Aufzählungstypen deklariert, müssen die Wertebereiche disjunkt sein. Zum Beispiel ist

(ROT, BLAU, GRUEN)
(GELB, ORANGE)
erlaubt, aber
(ROT, BLAU, GRUEN)
(ROT, GELB)

nicht, da der Wert ROT nicht zwei Datentypen angehören darf.
Intern werden Aufzählungstypen in einem Byte dargestellt. Dadurch ist die maximale Anzahl von Werten auf 256 beschränkt.

### 4.3. *Teilbereichstypen*

Teilbereichstypen ermöglichen die Eingrenzung des Wertebereiches von Datenobjekten auf einen Teil eines sogenannten Basistyps. Basistypen können sein INTEGER, BOOLEAN, CHAR und Aufzählungstypen. Die Deklaration zeigt das Syntaxdiagramm im Bild 4.3. Die Konstanten müssen von einem der erlaubten Basistypen sein.

→ *konstante* → .. → *konstante* →

*Bild 4.3 Syntaxdiagramm „teilbereichstyp"*

Es folgen einige Beispiele:

**Teilbereiche des Typs INTEGER**
**1...12**
**−3...3**
**0...100**
**Teilbereiche des Typs CHAR**
**'A'...'Z'**
**'0'...'9'**
Teilbereiche des Aufzählungstyps
**(Mo, Di, Mi, Don, Fr, Sa, So)**
**Mo...Fr**
**Sa...So**
**Di...Do**

Intern werden Teilbereichstypen wie ihr Basistyp dargestellt.
PASCAL-Systeme sichern die Einhaltung des Wertebereiches von Teilbereichstypen. Soll einem Datenobjekt ein nicht erlaubter Wert zugewiesen werden, wird die Abarbeitung des Programms mit einer entsprechenden Fehlermeldung beendet. Diese Überprüfung erhöht den Speicherplatzbedarf und die Laufzeit eines Programms. Deshalb kann sie durch die Compileroption ○R- bzw. ○R+ aus- bzw. eingeschaltet werden.

## 5. Struktur eines PASCAL-Programms

### 5.1. *Überblick*

Der grundlegende Baustein eines PASCAL-Programms ist ein sogenannter *Block*. Seine Syntax zeigt Bild 5.1. Ein Block besteht aus Deklarations- und Anweisungsteil. Mit ihm können Prozeduren und Funktionen sowie das Hauptprogramm selbst gebildet werden.

→ *deklarationsteil* → *anweisungsteil* →

*Bild 5.1 Syntaxdiagramm „block"*

Jedes PASCAL-Programm einschließlich aller benötigten Prozeduren und Funktionen ist eine einzige *Übersetzungseinheit*. Das ermöglicht dem Compiler eine vollständige Kontrolle der Konsistenz aller Programmteile.

### 5.2. *Deklarationsteil*
### 5.2.1. *Überblick*

Das Bild 5.2. zeigt die Syntax des Deklarationsteils. Je nachdem, ob im Syntaxdiagramm die entsprechende waagerechte Alternative gewählt wird oder nicht, kann er Marken-, Konstanten-, Typen-, Variablen- sowie Prozedur- und Funktionsdeklarationen enthalten.
Während Marken-, Konstanten-, Typen- und Variablendeklarationen höchstens einmal und dann in der genannten Reihenfolge auftreten, können Prozedur- und Funktionsdeklarationen mehrmals vorhanden sein.

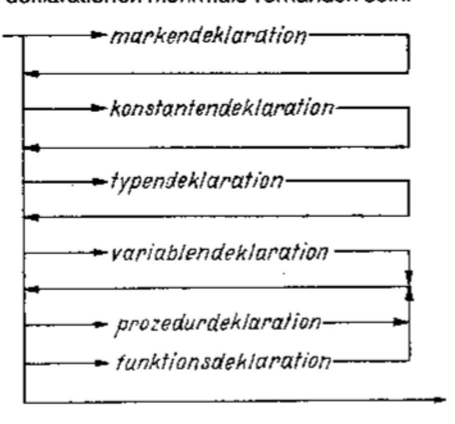



*Bild 5.2 Syntaxdiagramm „deklarationsteil"*

Das Syntaxdiagramm „deklarationsteil" läßt sich auch traversieren, ohne etwas zu deklarieren. Dann enthält der entsprechende Block keine Deklarationen.
In PASCAL herrscht *Deklarationszwang*. Alle Bezeichner müssen vor ihrer ersten Verwendung deklariert werden. Es gibt nur für den Datentyp Pointer eine Ausnahme. Bei TURBO-PASCAL ist die Syntax des Deklarationsteils geringfügig anders. Marken-, Konstanten-, Typen- und Variablendeklarationen dürfen mehrfach und auch gemischt auftreten. Bezeichner müssen aber auch dort vor ihrer Verwendung deklariert sein.
In den nachfolgenden Unterpunkten werden Marken-, Konstanten-, Typen- und Variablendeklarationen besprochen. Prozedur- und Funktionsdeklarationen werden solange aufgeschoben, bis der Anweisungsteil behandelt ist.

### 5.2.2. *Markendeklaration*

In PASCAL können beliebige Anweisungen mit einer Marke versehen werden. Mit der Goto-Anweisung kann dann unbedingt dorthin verzweigt werden.

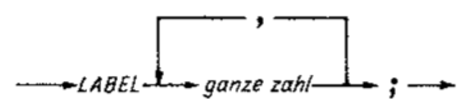



*Bild 5.3 Syntaxdiagramm „markendeklaration"*

Marken sind in PASCAL ganze Zahlen. Das Syntaxdiagramm in Bild 5.3 zeigt die Markendeklaration. Sie wird durch den reservierten Bezeichner LABEL eingeleitet. Beispiele sind

LABEL 99;
LABEL 10, 20, 30;

In TURBO-PASCAL können auch Bezeichner als Marken deklariert werden.

*(Fortsetzung folgt)*

# PASCAL

## (Teil 2)

Dr. Claus Kofer
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

### 5.2.3. Konstantendeklaration

Die Konstantendeklaration ermöglicht die Einführung von Bezeichnern für solche Datenobjekte, deren Wert von vornherein feststeht und sich auch während der Programmabarbeitung nicht ändert. Die Bezeichner können dann synonym für die entsprechende Konstante verwendet werden.




*Bild 5.4 Syntaxdiagramm „konstantendeklaration"*

Das Bild 5.4 zeigt die Syntax der Konstantendeklaration. Sie wird durch den reservierten Bezeichner CONST eingeleitet. Mit

CONST Epsilon = 0.01;

wird z. B. die Konstante Epsilon deklariert, die vom Typ REAL ist und den Wert 0.01 hat.
Es können auch mehrere Konstanten deklariert werden. Durch

```
CONST Max = 100;
      Delta = -1.5;
      Kontrolldruck = FALSE;
      Prompt = '>';
      Text = 'Programm V1.0';
```

werden die Konstanten

- Max vom Typ INTEGER mit dem Wert 100,
- Delta von Typ REAL mit dem Wert −1.5,
- Kontrolldruck vom Typ BOOLEAN mit dem Wert FALSE,
- Prompt vom Typ CHAR mit dem Wert '>' und
- Text vom Typ Zeichenkette mit dem Wert 'Programm V1.0'

eingeführt.

In Standard-PASCAL können nur INTEGER-, REAL-, BOOLEAN-, CHAR- und Zeichenkettenkonstanten deklariert werden.
Als Erweiterung gegenüber dem Standard gestattet TURBO-PASCAL die Deklaration getypter Konstanten. Der Typ ist mit Ausnahme von File beliebig, also z. B. auch strukturiert. Siehe dazu auch Punkt 11.2.

### 5.2.4. Typendeklaration

Mit der Typendeklaration werden Bezeichner für Datentypen eingeführt.

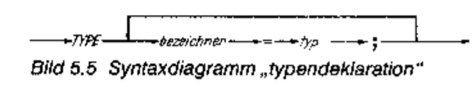



*Bild 5.5 Syntaxdiagramm „typendeklaration"*

Das Bild 5.5 zeigt die Syntax. Durch

TYPE Farbe = (ROT, BLAU, GRUEN);

wird z. B. der Typenbezeichner Farbe eingeführt. Der Bezeichner Farbe repräsentiert einen Aufzählungstyp und kann im nachfolgenden Programmtext ebenso verwendet werden wie der Bezeichner der Standardtypen INTEGER, REAL usw.

Weitere Beispiele für Typendeklarationen sind:

- TYPE Wochentag = (Mo,Di,Mi,Don,Fr, Sa,So);
- VglTyp = (kleiner,gleich,größer);
- EAFunktion = (Lesen, Schreiben, Positionieren);
- Sektorbereich = 1..12;
- Buchstabe = 'A'..'Z';
- Ziffer = '0'..'9'.

### 5.2.5. Variablendeklaration

Durch die Variablendeklaration werden Bezeichner für Variablen eingeführt. Gleichzeitig wird für jede Variable ihr Datentyp angegeben. Damit steht fest, welche Werte sie im Verlauf der Programmabarbeitung annehmen kann.

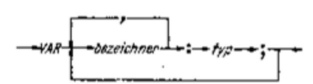



*Bild 5.6 Syntaxdiagramm „variablendeklaration"*

Das Bild 5.6 zeigt die Syntax. Die Variablendeklaration wird eingeleitet durch den reservierten Bezeichner VAR. Durch

```
VAR I : INTEGER;
    X : REAL;
    C : CHAR;
    B : BOOLEAN;
```

werden die Variablen I, X, C und B deklariert. Gleichzeitig wird aber auch festgelegt, daß I nur Werte aus dem Wertebereich von INTEGER annehmen kann, X aus dem Wertebereich von REAL usw.
Unter der Benutzung der weiter oben angeführten Aufzählung- und Teilbereichstypen ist folgende Variablendeklaration denkbar:

```
VAR Flagge : Farbe;
    Tag    : Wochentag;
    Sekt   : Sektorbereich;
```

Die variable Flagge ist vom Aufzählungstyp Farbe. Sie kann während der Programmabarbeitung nur einen der Werte ROT, BLAU oder GRUEN annehmen. Analog kann die Variable Tag nur einen der Werte Mo, Di, Mi, Don, Fr, Sa oder So annehmen. Der Typ der Variablen Sekt ist der Teilbereich 1...12 des Standardtyps INTEGER. Demzufolge sind ihre möglichen Werte eingeschränkt auf 1 ⇐ Sekt ⇐ 12.
Mehrere Variablen gleichen Typs können abkürzend in folgender Form deklariert werden:

```
VAR I,J,K     : INTEGER;
    X,Z,U,V,W : REAL;
```

In PASCAL ist es nicht möglich, den Variablen bei ihrer Deklaration einen Anfangswert zu geben.

### 5.3. *Anweisungsteil*

### 5.3.1. Überblick

Den Aufbau des Anweisungsteils zeigt Bild 5.7. Eine Liste von Anweisungen wird durch BEGIN und END geklammert. Das Semikolon ist das Trennzeichen zwischen den Anweisungen. Die einzelnen Anweisungen werden in Bild 5.8 gezeigt.

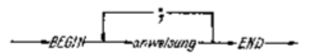



*Bild 5.7 Syntaxdiagramm „anweisungsteil"*

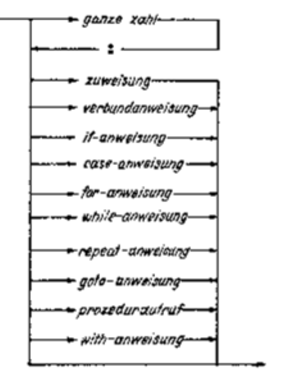



*Bild 5.8 Syntaxdiagramm „anweisung"*

Für den Anweisungsteil hat die syntaktische Einheit „ausdruck" eine gewisse zentrale Bedeutung. Sie wird im nächsten Unterpunkt besprochen. Daran schließen sich die einzelnen Anweisungen an. Prozeduraufruf und With-Anweisung werden jedoch aufgeschoben bis zu Prozeduren und Funktionen im Punkt 5.5. bzw. bis zum Datentyp Rekord im Punkt 8.

### 5.3.2. Ausdrücke

Die PASCAL-Notation eines Ausdruckes orientiert sich an der üblichen mathematischen Schreibweise. Seine Deklaration erfolgt mit Hilfe der Syntaxdiagramme „einfacher ausdruck", „term" und „faktor". Sie werden in den Bildern 5.9 bis 5.12 gezeigt.

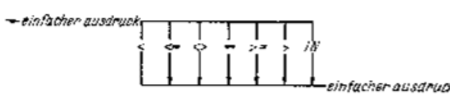



*Bild 5.9 Syntaxdiagramm „ausdruck"*

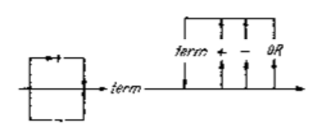



*Bild 5.10 Syntaxdiagramm „einfacher ausdruck"*

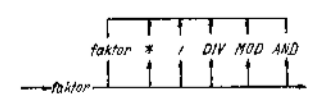



*Bild 5.11 Syntaxdiagramm „term“*

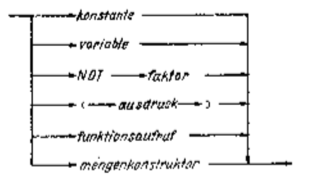



*Bild 5.12 Syntaxdiagramm „faktor“*

Faktor enthält Alternativen, die eine Bezugnahme auf die Datenobjekte des PASCAL-Programms gestatten.
Durch die hierarchische Deklaration von Ausdruck erhalten die Operatoren unterschiedliche Prioritäten. Die nachfolgende Aufstellung zeigt sie nach fallender Priorität angeordnet:

( ) NOT
* / DIV MOD AND
\+ – OR
< <= < > = >= > IN

Operatoren einer Zeile haben die gleiche Priorität.
Der Compiler weist Ausdrücke zurück, in denen auf Datenobjekte Operatoren angewendet werden, die für diese nicht erklärt sind. Dabei dürfen auf Teilbereichstypen grundsätzlich dieselben Operatoren angewendet werden, wie auf den entsprechenden Basistyp.
Als einzige Ausnahme organisiert der Compiler eine ggf. erforderliche Umwandlung von INTEGER in REAL.
Beispiele für korrekte Ausdrücke sind:

```
I+1
0.5*X+K
'Eine Zeichenkette'
I<100
Error OR PRINT AND LIST
(I<0) OR (100<I)
```

Im letzten Ausdruck sind die Klammern wegen der höheren Priorität von OR notwendig.
Nicht korrekte Ausdrücke sind:

```
'A' + 1
I AND 255
```

### 5.3.3. Zuweisung

Eine Zuweisung gibt einer Variablen einen neuen Wert. Das Bild 5.14 zeigt die Syntax. Die Zeichenkombination := ist der Zuweisungsoperator.

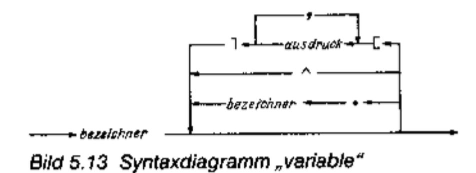
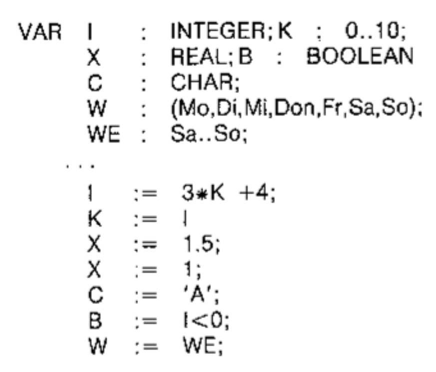



*Bild 5.13 Syntaxdiagramm „variable“*

variable := ausdruck

*Bild 5.14 Syntaxdiagramm „zuweisung“*

Der Compiler lehnt Zuweisungen ab, bei denen der Typ des Ausdruckes nicht mit dem der Variablen verträglich ist. Typen sind miteinander verträglich, wenn sie durch den gleichen Typenbezeichner deklariert wurden. Teilbereichstypen sind mit ihrem Basistyp verträglich. Als einzige Ausnahme organisiert der Compiler die Konvertierung von INTEGER in REAL.
Das nachfolgende Programmstück zeigt korrekte Zuweisungen:



```
VAR I  : INTEGER; K  : 0..10;
    X  : REAL; B  : BOOLEAN
    C  : CHAR;
    W  : (Mo,Di,Mi,Don,Fr,Sa,So);
    WE : Sa..So;
  ...
    I  := 3*K +4;
    K  := I
    X  := 1.5;
    X  := 1;
    C  := 'A';
    B  := I<0;
    W  := WE;
```

### 5.3.4. Verbundanweisung

Die Verbundanweisung hat ausschließlich eine syntaktische Funktion. Sie läßt eine Liste von Anweisungen wie eine einzige Anweisung wirken.

BEGIN anweisung ; END

*Bild 5.15 Syntaxdiagramm „verbundanweisung“*

Die Syntax ist in Bild 5.15 dargestellt. Ein Beispiel ist:

```
BEGIN I:=1; VOLL:=TRUE; X:=0.5 END
```

### 5.3.5. If-Anweisung

Die If-Anweisung bewirkt eine Verzweigung im Programmablauf. Das Bild 5.16 zeigt die Syntax. Die auf die reservierten Bezeichner THEN bzw. ELSE folgenden Anweisungen heißen THEN- bzw. ELSE-Zweig.

IF ausdruck THEN anweisung ELSE anweisung

*Bild 5.16 Syntaxdiagramm „if-anweisung“*

Bei der Abarbeitung der If-Anweisung wird zuerst der Ausdruck ausgewertet. Er muß einen Wert vom Typ BOOLEAN liefern. Ist er TRUE, wird der THEN-Zweig durchlaufen, sonst der ELSE-Zweig.

Beispiele:

```
IF X<>0 THEN Y:= 1/X ELSE Y:=1.E38
IF A>B THEN MAX:=A ELSE MAX:=B
```

Durch Nutzung der Verbundanweisung können THEN- und ELSE-Zweig durch weitere Anweisungen aufgefüllt werden:

```
IF I > MAX THEN BEGIN
   VOLL := TRUE; I:=0
END ELSE BEGIN
   K:= I DIV 4; I:=I+1; X:=1.0
END
```

Die Syntax läßt es zu, daß der gesamte ELSE-Zweig fehlen darf:

```
IF X < 0 THEN X:= –X
```

Es ist zulässig, If-Anweisungen beliebig ineinander zu verschachteln. Entstehende Mehrdeutigkeiten werden durch die Festlegung beseitigt, daß ein ELSE-Zweig stets zur letzten IF-Anweisung gehört:

```
IF a1 THEN
   IF a2 THEN anw2
      ELSE anw3
```

entspricht

```
IF a1 THEN BEGIN
   IF a2 THEN anw2 ELSE anw3
END
```

Soll der ELSE-Zweig zur ersten IF-Anweisung gehören, ist zu schreiben

```
IF a1 THEN BEGIN
   IF a2 THEN anw2 END
ELSE anw3
```

### 5.3.6. Case-Anweisung

Die Case-Anweisung ist eine ***Mehrwegverzweigung***. Ihre Struktur wird im Bild 5.17 gezeigt. Sie besteht aus einem Ausdruck und einer Liste, die durch Konstanten und Anweisungen gebildet wird. Die Konstanten und der Ausdruck müssen den gleichen einfachen Typ haben. REAL ist jedoch nicht zugelassen.

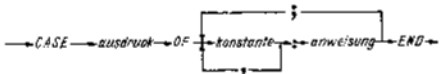



*Bild 5.17 Syntaxdiagramm „case-anweisung“*

Die Abarbeitung der Case-Anweisung beginnt mit der Berechnung des Ausdruckes. Dann wird die Anweisung abgearbeitet, deren vorangestellte Konstante der Wert des Ausdruckes ist, und danach die Case-Anweisung verlassen. Gibt es keine solche Anweisung, wird die Programmabarbeitung bei der nachfolgenden Anweisung fortgesetzt.
Es folgt ein Beispiel mit dem Aufzählungstyp Wochentag:

```
TYPE  Wochentag=(Mo,Di,
           Mi,Don,Fr,Sa,So)
VAR   Tag : Wochentag;
.
.
.
CASE Tag OF
  Mo : Anweisung 1;
  Fr : Anweisung 2;
  Sa : Anweisung 3;
END
```

Falls die Variable Tag den Wert Mo hat, wird Anweisung 1 abgearbeitet und danach die Case-Anweisung verlassen; falls Tag den Wert Fr hat, wird Anweisung 2 abgearbeitet und danach die Case-Anweisung verlassen usw. Hat die Variable Tag keinen der Werte Mo, Fr oder Sa, wird die Case-Anweisung sofort verlassen.
Ist für mehrere Konstanten die erforderliche Aktion gleich, können diese Konstanten zu einer Liste zusammengefaßt werden. Das nachfolgende Programmstück demonstriert das am Beispiel der Klassifikation von Operatoren.

```
TYPE OpTyp = (AddOp, Mulop);
VAR Operator : Op Typ;
      C : CHAR;
.
.
.
CASE C OF
'+','-' : Operator := AddOp;
'*','/' : Operator := MulOp
END
```

Falls die Variable C den Wert '+' oder '−'hat, wird der Variablen Operator der Wert AddOp zugewiesen. Ist der Wert von C '*'oder '/', dann erhält Operator den Wert MulOp. Es wäre sicher noch wünschenswert, auch die Operatoren OR, DIV, MOD und AND zu klassifizieren. Das ist mit diesem einfachen Programmstück aber nicht möglich. Als Konstanten in der Case-Anweisung dürfen keine Zeichenketten stehen.
TURBO-PASCAL läßt bei der Case-Anweisung einen ELSE-Zweig zu. Er wird durchlaufen, falls der Wert des berechneten Ausdrukkes mit keiner Konstanten übereinstimmt. Zur Demonstration wird das vorangegangene Beispiel erweitert.

```
TYPE OpTyp = (AddOp, MulOp, NoOp);
VAR  Operator : OpTyp;
     C : CHAR;
   .....
CASE C OF
     '+','-' : Operator := AddOp;
     '*','/' : Operator := MulOp
     ELSE      Operator := NoOP
END
```

Die Variable Operator wird mit dem Wert NoOP belegt, falls C keinen der Werte '+', '−', '*' oder '/' hat.
Die Syntax der Case-Anweisung läßt vor ELSE und END kein Semikolon zu.
TURBO-PASCAL bietet eine weitere Vereinfachung: Die einer Anweisung vorangehende Liste von Konstanten darf auch eine Teilbereichsangabe sein. Damit darf anstelle von

```
CASE c OF
'A','B','C','D','E','F','G': anweisung;
 ...
```

einfacher geschrieben werden

```
CASE c OF
'A'..'G': anweisung;
```

### 5.3.7. For-Anweisung

Die For-Anweisung stellt eine *Schleifenkonstruktion* dar. Bild 5.18 zeigt die Syntax. Die Variable muß vom einfachen Typ sein. REAL ist jedoch nicht erlaubt.
Die Ausdrücke dienen zur Angabe von Anfangs- und Endwerten, zwischen denen die Variable in ansteigender (TO) bzw. abfallender (DOWNTO) Folge Werte annimmt. Eine Wahl der Schrittweite ist nicht möglich.
Die Ausdrücke werden einmal zu Beginn berechnet. Die Schleifenbedingung wird vor jeder Abarbeitung der Schleifenanweisung getestet.
Beispiele für For-Anweisungen sind

```
FOR I:= 1 TO Max DO Fak:=Fak*I
FOR I:= Max DOWNTO 1 DO S:=S+1/I
FOR C:= 'A' TO 'Z' DO ...;
FOR C:= '9' DOWNTO '0' DO ...;
FOR Tag:= Mo TO Sa DO ...;
FOR B:= FALSE TO TRUE DO ...;
```

Sollen bei einem Schleifendurchlauf mehrere Anweisungen abgearbeitet werden, sind sie zu einer Verbundanweisung zu klammern:

```
FOR I:=1 TO MAX DO BEGIN
    S:= S+Q; Q:= Q*Q/I
END
```

### 5.3.8. While-Anweisung

Die While-Anweisung stellt ebenfalls eine *Schleifenkonstruktion* dar. Die Syntax wird in Bild 5.19 gezeigt. Der Ausdruck muß einen Wert vom Typ BOOLEAN liefern.

→WHILE → ausdruck → DO → anweisung→

*Bild 5.19 Syntaxdiagramm „while-anweisung"*

Bei der While-Anweisung wird vor jedem Schleifendurchlauf der Ausdruck ausgewertet. Liefert er den Wert FALSE, wird die Schleife verlassen.
Das nachfolgende Programmstück zeigt die Anwendung der While-Anweisung zur Berechnung der Summe S=1/N+...+1/2+1.

```
S:=0; I:=N;
WHILE I>0 DO BEGIN
    S:=S+1/I; I:=I-1
END
```

### 5.3.9. Repeat-Anweisung

Die Repeat-Anweisung stellt eine weitere Schleifenkonstruktion dar (Syntax siehe Bild 5.20).

→REPEAT → anweisung (;) → UNTIL → ausdruck→

*Bild 5.20 Syntaxdiagramm „repeat-anweisung"*

Eine Liste von Anweisungen wird durch die reservierten Bezeichner REPEAT und UNTIL geklammert.
Bei jedem Schleifendurchlauf wird zuerst die Anweisungsliste abgearbeitet und danach der Ausdruck berechnet. Der Ausdruck muß einen Wert von Typ BOOLEAN liefern. Ist sein Wert TRUE, wird die Repeat-Anweisung verlassen.
Als Beispiel für die Anwendung der Repeat-Anweisung wird wieder die Berechnung der Summe S=1/N+...+1/2+1 gezeigt:

```
S:=0; I:=N
REPEAT
   S:=S+1/I; I:=I-1
UNTIL I=0
```

Da die Anweisungsliste wenigstens einmal abgearbeitet wird, führt das Programm für N=0 zu einem Fehler.

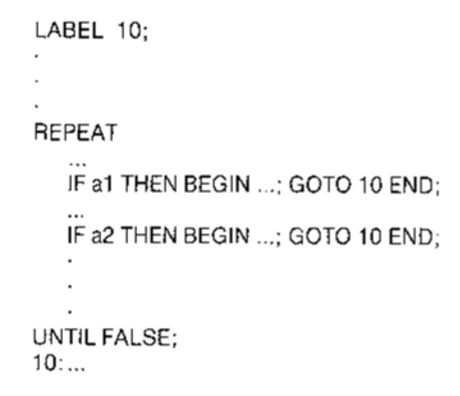
### 5.3.10. Goto-Anweisung

Das Bild 5.21 zeigt die Syntax der Goto-Anweisung. Sie bewirkt eine unbedingte Verzweigung zu der Anweisung mit der angegebenen Marke.

→GOTO → ganze zahl→

*Bild 5.21 Syntaxdiagramm „goto-anweisung"*

Das nachfolgende Programmbeispiel demonstriert, daß die PASCAL-Anweisungen zur Steuerung des Programmablaufs durchaus sinnvoll durch die Goto-Anweisung ergänzt werden können. Es wird eine Schleife mit verschiedenen Ausgängen und differenzierter Nachbehandlung gezeigt:



```
LABEL 10;
.
.
.
REPEAT
    ...
    IF a1 THEN BEGIN ...; GOTO 10 END;
    ...
    IF a2 THEN BEGIN ...; GOTO 10 END;
    .
    .
    .
UNTIL FALSE;
10:...
```

Bei TURBO-PASCAL dürfen Sprünge nicht aus einem Block herausführen.

### *5.4. Hauptprogramm*

Ein PASCAL-Hauptprogramm besteht aus einem *Programmkopf* und dem Programmbaustein Block. Bild 5.22 zeigt dies. Die Syntax des Programmkopfes zeigt Bild 5.23. Er wird durch den reservierten Bezeichner PROGRAM eingeleitet. Es folgt der Name des Programms. Er kann für eine inhaltliche Kennzeichnung des Programms verwendet werden, hat darüber hinaus aber keine Bedeutung.

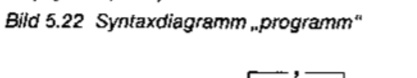



*Bild 5.22 Syntaxdiagramm „programm"*

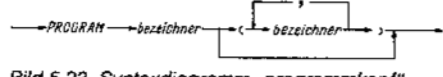



*Bild 5.23 Syntaxdiagramm „programmkopf"*

Dem Programmnamen kann eine in runde Klammern eingeschlossene Liste von Bezeichnern folgen. Sie stellen die Verbindung des PASCAL-Programms zum sogenannten Environment dar. In der Regel sind das Dateien, mit denen das Programm arbeiten kann, ohne sie eröffnen oder anlegen zu

→FOR → variable → := → ausdruck → TO / DOWNTO → ausdruck → DO → anweisung→

*Bild 5.18 Syntaxdiagramm „for-anweisung"*

müssen. In TURBO-PASCAL hat diese Liste keine Bedeutung. Abgeschlossen wird ein PASCAL-Programm durch einen Punkt.
Der Programmkopf des Beispiels aus Punkt 2.7 ist demnach

```
PROGRAM Tabelle (INPUT,OUTPUT)
```

Das Programm heißt Tabelle. Das benutzte Environment sind die Dateien INPUT und OUTPUT.
Es folgt ein Block, bei dem der Deklarationsteil nur die Variablendeklaration

```
VAR x,y,max:INTEGER;
```

enthält. Der folgende Anweisungsteil

```
BEGIN
  Readln(INPUT,max);
  FOR x:=0 TO max DO BEGIN
    y:=x*x; Writeln(OUTPUT,
    'x:=', x,'y:=', y)
  END
END
```

besteht aus einer Liste von zwei Anweisungen, einem noch zu besprechenden Prozeduraufruf und einer For-Anweisung. Das Schleifeninnere ist eine Verbundanweisung. Der Punkt beendet das Programm und damit einen Satz in der Programmiersprache PASCAL.

## 5.5. *Prozeduren und Funktionen*

### 5.5.1. Einführung

Prozeduren und Funktionen gestatten die Untergliederung eines Programms in relativ selbständige Teile. Das ursprüngliche Motiv war die Einsparung von Programmkode. Gleiche Programmpassagen werden durch Aufrufe einer Prozedur ersetzt, welche die erforderlichen Anweisungen enthält.
Zunehmend trat aber ein anderer Gesichtspunkt in den Vordergrund: Prozeduren und Funktionen gestatten es, inhaltlich zusammengehörende und eine gewisse abgeschlossene Leistung erbringende Programmteile im Programmtext als solche sichtbar zu machen. Damit läßt sich die Übersichtlichkeit eines Programms wesentlich verbessern.
In den nachfolgenden Unterpunkten wird zunächst die Syntax von Deklaration und Aufruf gezeigt. Danach werden Gültigkeitsbereiche von Bezeichnern und Speicherplatzzuordnung für lokale Variablen besprochen. Den Abschluß bilden Standardprozeduren und -funktionen.

### 3.5.2. Deklaration und Anruf

Ähnlich wie das Hauptprogramm werden Prozeduren und Funktionen durch den Kopf und den Programmbaustein Block gebildet. Die Bilder 5.24 und 5.26 zeigen das. Die an den reservierten Bezeichner FORWARD geknüpfte Alternative wird später behandelt.

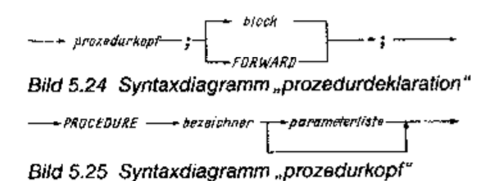



*Bild 5.24 Syntaxdiagramm „prozedurdeklaration"*

*Bild 5.25 Syntaxdiagramm „prozedurkopf"*




*Bild 5.26 Syntaxdiagramm „funktionsdeklaration"*

Bei Prozeduren wird der Kopf durch den reservierten Bezeichner PROCEDURE eingeleitet. Ihm folgt der Name der Prozedur. Wahlweise kann eine Parameterliste angegeben werden. Die Syntaxdiagramme 5.24 und 5.25 zeigen dies.
Bei Funktionen trägt der Funktionsname selbst einen Wert. Sein Typ wird, wie das Bild 5.27 zeigt, der Parameterliste nachgestellt. Erlaubt sind nur einfache Typen.

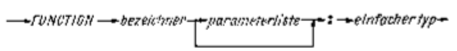



*Bild 5.27 Syntaxdiagramm „funktionskopf"*

Eine genaue Betrachtung der Syntax von Block zeigt, daß der Deklarationsteil weitere Prozeduren und Funktionen enthalten kann. Die Diskussion aller damit zusammenhängenden Probleme wird jedoch bis zum nächsten Unterpunkt aufgeschoben.
Eine Prozedur kann über sogenannte globale Variablen und Parameter mit ihrer Umwelt kommunizieren. Die Parameter werden in einer Parameterliste angegeben. Die einzelnen Elemente dieser Liste heißen bei der Deklaration formale Parameter, beim Aufruf aktuelle Parameter.

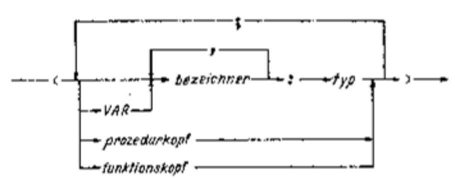



*Bild 5.28 Syntaxdiagramm „parameterliste"*

Das Bild 5.28 zeigt die Syntax der Parameterliste. Die einzelnen Alternativen repräsentieren die vier möglichen Parametertypen:

- Werteparameter
- Referenzparameter
- Prozedurparameter und
- Funktionsparameter.

Werte- und Referenzparameter übermitteln Datenobjekte. Ihnen wird deshalb eine Typangabe nachgestellt.
Während Werteparameter nur die Übertragung von Informationen an die Prozedur oder Funktionen gestatten, wirken Referenzparameter in beiden Richtungen.
Die Aktivierung von Prozeduren und Funktionen zeigen die Syntaxdiagramme der Bilder 5.29 und 5.30. Beide Formen sind identisch. Während aber der Prozeduraufruf eine selbständige Anweisung ist, können Funktionen nur als Faktor in einen Ausdruck aufgerufen werden.

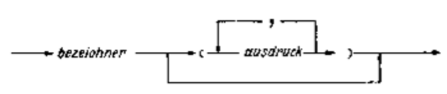



*Bild 5.29 Syntaxdiagramm „prozeduraufruf"*

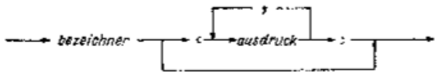



*Bild 5.30 Syntaxdiagramm „funktionsaufruf"*

Der Compiler löst die Aufrufe so auf, daß zuerst die Ausdrücke ausgewertet werden und danach in die entsprechende Prozedur oder Funktion verzweigt wird.
Vom Compiler wird überprüft, ob

- Anzahl und Typ der aktuellen Parameter mit den formalen übereinstimmen
- ein aktueller Referenzparameter eine Variable ist und eine
- Übereinstimmung zwischen den Parameterlisten der aktuellen und formalen Prozedur- und Funktionsparameter herrscht.

Es folgen Beispiele:
Die Prozedur Max 2 soll von zwei Parametern den Wert des größeren ermitteln und zurückgeben.

```
PROCEDURE Max2
        (A,B:REAL;VAR Max:REAL);
BEGIN
   IF A>B THEN Max:=A ELSE-
 Max:=B
END.
```

Im Prozedurkopf werden Name und Parameterliste angegeben. Die Parameterliste enthält die Werteparameter A und B vom Typ REAL und den Referenzparameter Max, der ebenfalls vom Typ REAL ist. Der Deklarationsteil ist leer. Es werden zur Ermittlung des Resultates keine Marken, Konstanten, Typen und Variablen benötigt. Der Anweisungsteil besteht aus einer If-Anweisung. Da der Parameter Max Informationen aus der Prozedur zurückübermitteln soll, muß er Referenzparameter sein.
Die Prozedur Max2 läßt sich nun wie folgt aufrufen:

```
Max2 (x,y,z)
Max2 (1.5*x,y,z)
Max2 (1.5*x,3,z)
```

Der dritte Parameter von Max2 muß stets eine Variable sein.
Die Prozedur Max2 läßt sich auch als Funktion umschreiben:

```
FUNCTION Max2 (A,B:REAL):REAL;
BEGIN
   IF A>B THEN Max2:= A
                  ELSE Max2:= B
END;
```

Im Funktionskopf werden wieder Name und Parameterliste angegeben. Der Parameter Max fehlt hier. Seine Aufgabe übernimmt der Funktionsname. Die mit dem Doppelpunkt angefügte Typenangabe erklärt ihn zum Datenobjekt vom Typ REAL.

***wird fortgesetzt***

*Hinweis:* In Bild 2.6 des ersten Teils unserer PASCAL-Folge (MP 9/87) sind an Stelle von „T" Apostrophe zu setzen. *Red.*

# PASCAL (Teil 3)

**Dr. Claus Kofer**
**Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden**

Der Deklarationsteil der Funktion ist wieder leer. In der If-Anweisung des Anweisungsteils wird dem Funktionsnamen ein Wert zugewiesen. Auf diesen Wert kann durch Anruf der Funktion Max2 Bezug genommen werden. Zum Beispiel in folgender Form:

```
Z:=Max2(x,y)
Z:=Max2(1.0,2.5*y)
IF Z>Max2 (1.5*x,y) THEN ...
REPEAT ... UNTIL Max2(x,y)<1.0
```

Es wird jetzt eine Prozedur Swap gezeigt, die die Werte ihrer Parameter vertauscht.

```
PROCEDURE Swap (VAR A,B:CHAR);
VAR H:CHAR
BEGIN
  H:=A;   A:=B;   B:=H
END;
```

Im Prozedurkopf werden wieder Name und formale Parameter angegeben. Es sind beides Referenzparameter. Die verwendete Schreibweise ist eine Abkürzung für PROCEDURE Swap (VAR A:CHAR; VAR B:CHAR).
Da das Vertauschen ohne eine Hilfsvariable nicht möglich ist, wird diese im Deklarationsteil der Prozedur Swap deklariert. Der Anweisungsteil besteht aus einer Liste von drei Zuweisungen, die das Vertauschen bewerkstelligen.
Die aktuellen Parameter der Prozedur Swap dürfen nur Variablen vom Typ CHAR sein:

```
Swap (C1,C2)
```

Aber nicht

```
Swap ('A',C1)
```

Für die Verwendung von Funktionsparametern wird nun ein Beispiel angegeben:

```
FUNCTION Diff(XO,DX:REAL;
         Fkt(x:REAL):REAL):REAL;
BEGIN
  Diff:=(Fkt(XO+DX)-Fkt(XO))/DX
END;
```

Der Funktionskopf von Diff enthält neben der bereits bekannten Deklaration der Werteparameter XO und DX einen formalen Funktionsparameter Fkt. Laut Deklaration ist Fkt eine Funktion mit einem Parameter vom Typ REAL und liefert selbst einen Funktionswert vom Typ REAL. Im Anweisungsteil wird der Wert der Funktion Diff als Differenzenquotient der Funktion Fkt für das Argument XO berechnet.

Als aktueller Parameter darf für Fkt jede beliebige Funktion verwendet werden, die einen Parameter vom Typ REAL hat und selbst einen Wert vom Typ REAL liefert. Es werden zwei solcher Funktionen gezeigt:

```
FUNCTION P1(x:REAL):REAL;
BEGIN
  P1:= (1.5*x+2.5)*x+3.5
END;

FUNCTION P2(x:REAL):REAL
BEGIN
  P2:=1/((0.5*x+0.25)*x
END;
```

Unter Verwendung von P1 und P2 sind nun folgende Aufrufe der Funktion Diff möglich

```
A:=Diff(x,0.01,P1);
A:=Diff(x,0.01,P2)
```

Leider gibt es bei den meisten PASCAL-Systemen für Prozedur- und Funktionsparameter die Einschränkung, daß als aktuelle Parameter nicht die sogenannten Standardprozeduren und -funktionen verwendet werden dürfen.
Bei TURBO-PASCAL sind Prozedur- und Funktionsparameter überhaupt nicht implementiert.

*5.5.3. Blockstruktur, Gültigkeitsbereiche von Bezeichnern*

Nach den Syntaxdiagrammen der Bilder 5.1, 5.2, 5.24 und 5.26 können Prozeduren und Funktionen selbst wieder weitere Prozeduren und Funktionen enthalten. Es entsteht eine Struktur von ineinander verschachtelten Blöcken, von denen jeder aus einem (möglicherweise leeren) Deklarationsteil und einem Anweisungsteil besteht.
Bild 5.31 zeigt eine typische Blockstruktur. Die Prozeduren A und C sind Bestandteil der Deklarationen des Hauptprogramms. Der Deklarationsteil der Prozedur A enthält die Prozedur B.
Es sind beliebige Schachtelungen von Prozeduren und Funktionen denkbar, allerdings begrenzen die PASCAL-Systeme oft deren Tiefe. Im Bild 5.31 ist die größte Schachtelungstiefe drei: Hauptprogramm, Prozeduren A und B.
In jedem Block können Marken, Konstanten, Datentypen, Variablen sowie weitere Prozeduren und Funktionen deklariert werden. Das wirft die Frage nach den Gültigkeitsbereichen der Bezeichner auf. Es gibt dafür zwei einfache Regeln:
1. Ein Bezeichner gilt im Inneren des gesamten Blocks, in dem er deklariert wurde.
2. Bei gleichen Bezeichnern gilt die lokalere Deklaration.
Der Name einer Prozedur bzw. Funktion gehört zu den Bezeichnern des umschließenden Blocks. Die möglicherweise vorhandenen formalen Parameter sind jedoch als Bestandteil des Deklarationsteils der Prozedur bzw. Funktion anzusehen.
Die Gültigkeitsbereiche der Bezeichner des Programmfragments aus Bild 5.31 werden tabellarisch im Bild 5.32 gezeigt. Die Bezeichner T und A des Hauptprogramms gelten in allen Blöcken. K gilt nur in den Blöcken PROG und C. In den Blöcken A und B wird K durch

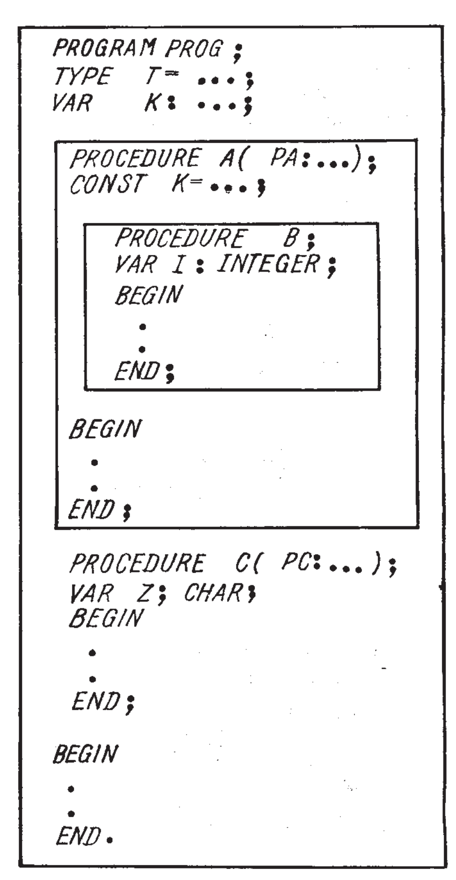



*Bild 5.31 Blockstruktur eines Programms*

| | | PROG | A | B | C |
|---|---|---|---|---|---|
| PROG | T | (1) | (1) | (1) | (1) |
| | K | (1) | | | (1) |
| | A | (1) | (1) | (1) | (1) |
| | C | (1) | | | (1) |
| A | PA | | (1) | (1) | |
| | K | | (1,2) | (1,2) | |
| | B | | (1) | (1) | |
| B | I | | | (1) | |
| C | PC | | | (1) | |
| | Z | | | (1) | |

*Bild 5.32 Gültigkeitsbereiche der Bezeichner aus Bild 5.31. Die Zeilen korrespondieren mit den Bezeichnern, die Spalten mit den Blöcken. Eine Ziffer zeigt an, aufgrund welcher Regel ein Bezeichner in einem Block gilt.*

eine Konstante mit dem gleichen Bezeichner überdeckt.
Der Bezeichner C gilt nur in den Blöcken PROG und C. In den Blöcken A und B ist C nicht bekannt, weil C noch nicht deklariert ist.
Die Bezeichner PA, K und B der Prozedur A gelten in den Blöcken A und B. Außerhalb des Blockes A sind sie nicht bekannt.
Der Bezeichner I der Prozedur B gilt nur im Block B.
Die Bezeichner PC und Z der Prozedur C gelten im Block C. Bild 5.31 zeigt, daß im Block C auch der Bezeichner A bekannt ist, denn seine Deklaration erfolgte im Programmtext schon vor dem Block C.
Für jeden Block können lokale und globale Deklarationen unterschieden werden. Lokale Deklarationen werden durch den Block selbst eingebracht. Globale Deklarationen stammen aus einschließenden Blöcken.
Die global deklarierten Variablen stellen eine weitere Möglichkeit zum Datenaustausch zwischen Prozedur oder Funktion und ihrer Umgebung dar. Das wird an folgendem Programmstück gezeigt:

```
PROGRAM M;
VAR x:REAL;
PROCEDURE A (PA:REAL);
BEGIN
  x:=PA;
END;
BEGIN
  A(1.0)
END.
```

In der Prozedur A ist neben dem formalen Parameter PA auch die globale Variable x bekannt. Sie darf demzufolge in den Anweisungen verwendet werden. Im Beispiel wird ihr der Wert des Parameters zugewiesen.
Der Aufruf A(1.0) läßt nicht erkennen, daß die Hauptprogrammvariable x manipuliert wurde. Die Nützlichkeit solcher „Seiteneffekte" soll hier nicht beurteilt werden. Der Programmierer muß wissen, daß es sie gibt und daß sie sich durch keine Compileroperation ausschalten lassen.

*5.5.4. Speicherplatzzuordnung für lokale Variablen*

Die durch die Blockstruktur eines PASCAL-Programms festgelegten Gültigkeitsbereiche der Bezeichner führen dazu, daß bei der Programmabarbeitung höchstens die Datenobjekte der aktiven Blöcke, d. h. des Hauptprogramms sowie der gerufenen Prozeduren und Funktionen, manipuliert werden können.
Da Speicherplatz immer eine kostbare Ressource ist, liegt es nahe, nur für Datenobjekte der aktiven Blöcke Speicherplatz bereitzustellen. Weil es sich aber erst zur Programmlaufzeit herausstellt, welche Blöcke aktiv sind, wird diese Strategie als dynamische Speicherplatzzuordnung bezeichnet.
Die Speicherpositionen der Datenobjekte können vom Compiler nur für das Hauptprogramm im voraus berechnet werden.
Die der Prozeduren und Funktionen ergeben sich in Abhängigkeit von ihrer konkreten Aufruffolge. Notwendige Adreßrechnungen müssen bis zur Laufzeit aufgeschoben werden. Die Maschinenbefehlssysteme moderner Rechner unterstützen solche Adreßrechnungen sehr gut, so daß PASCAL-Programme nur geringfügige Laufzeitnachteile gegenüber Programmen mit statischer Speicherplatzzuordnung wie z. B. FORTRAN haben.
Für die Programmierung ergeben sich aus der dynamischen Speicherplatzzuordnung folgende Konsequenzen:

– Der Wert von lokalen Variablen der Prozeduren und Funktionen ist stets als unbestimmt anzusehen, auch falls die Prozedur oder Funktion schon einmal gerufen wurde.
– Rekursive Aufrufe von Prozeduren und Funktionen sind uneingeschränkt möglich.

Das Prinzip wird noch einmal mit Hilfe des Programmrumpfes von Bild 5.31 verdeutlicht. Das Bild 5.33a zeigt den belegten Datenspeicher für die Aufruffolge: Hauptprogramm, Prozedur A, Prozedur B. Beim Verlassen der Prozeduren B und A wird der belegte Datenspeicher wieder freigegeben. Da Prozeduren stets in der umgekehrten Reihenfolge ihres Aufrufs verlassen werden, wird der Datenspeicher stackartig verwaltet. Er wächst zumeist in Richtung fallender Adressen.

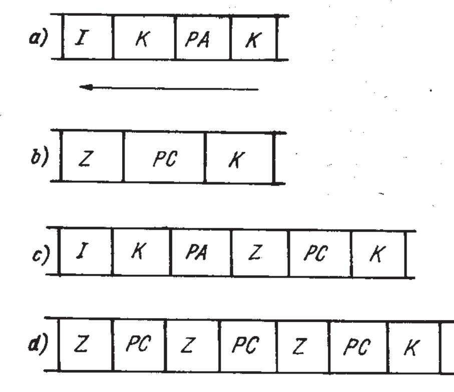


*Bild 5.33 Zuordnung von Speicher zu Datenobjekten der aktiven Prozeduren*

Die Bilder 5.33b und 5.33c zeigen weitere Situationen. Deutlich tritt auch hervor, daß die Position des Parameters PA und der lokalen Variablen I von der konkreten Aufruffolge abhängt – also erst zur Laufzeit feststeht.
Bild 5.33d zeigt schließlich die Struktur des Datenspeichers bei rekursiven Prozeduraufrufen. C wird vom Hauptprogramm aktiviert und ruft sich dann selbst wieder auf. Bei jedem Betreten von C wird ein neuer Bereich für lokale Variablen angelegt.
TURBO-PASCAL verfährt anders. Der Compiler ordnet den Datenobjekten in der Reihenfolge ihrer Deklaration statisch Platz im Datenspeicherbereich zu. Zur Programmabarbeitungszeit hat jedes Datenobjekt seinen festen Platz im Hauptspeicher. Rekursive Prozeduraufrufe führen so i. allg. zu Fehlern.
Der Vorteil dieser nicht PASCAL-gerechten Strategie liegt darin, daß auch bei Rechnern auf der Basis von 8-Bit-Prozessoren der Zugriff auf Datenobjekte sehr schnell ist und wenig Programmkode erfordert.
Rekursive Prozeduren werden nun in TURBO-PASCAL dadurch möglich, daß beim Betreten einer Prozedur zunächst die „alten" Werte der lokalen Variablen in einem Stack abgelegt und vor Verlassen der Prozedur rückgespeichert werden. Die Erzeugung der dafür erforderlichen Blocktransportbefehle wird durch die Compileroption ○A– und ○A+ ein- bzw. ausgeschaltet.

*5.5.5. Vorwärtsdeklaration*

Aufgrund des Deklarationszwanges muß ein Bezeichner vor seiner Benutzung deklariert werden. Prozeduren und Funktionen machen hier keine Ausnahme.
Es gibt aber durchaus sinnvolle Programme, bei denen diese Bedingung aufgrund des verwendeten Algorithmus nicht erfüllt werden kann. Das nachfolgende Programmstück verdeutlicht dies:

```
PROCEDURE A(PA:...);
BEGIN
  ...B(x);...
END;
PROCEDURE B(PB:...);
BEGIN
  ...A(y);...
END;
```

Der Aufruf von B im Anweisungsteil von A kann durch den Compiler nicht aufgelöst werden, da bis dorthin B unbekannt ist. Ein Vertauschen der Reihenfolge der Prozeduren A und B führt auch nicht zum Ziel.
Die Lösung erfolgt durch Trennung von Prozedurkopf und dazugehörendem Block mit Hilfe von FORWARD:

```
PROCEDURE B(PB:...);FORWARD;
PROCEDURE A(PA:...);
BEGIN
  ...B(x);...
END;
PROCEDURE B;
BEGIN
  ...A(y);...
END;
```

Innerhalb von A kann nun B aufgerufen werden. Durch die vorangegangene FORWARD-Deklaration ist der Kopf von B bekannt. Bei der Angabe des zu B gehörenden Blockes entfällt im Kopf die Parameterliste.

*5.5.6. Standardprozeduren und -funktionen*

Dem PASCAL-Compiler ist eine Reihe von Prozeduren und Funktionen bereits bekannt. Diese lassen sich in folgende Gruppen einteilen:

1. arithmetische Funktionen
2. Prozeduren und Funktionen zur Ein- und Ausgabe
3. Prozeduren zur Arbeit mit dynamischen Variablen.

Die Behandlung der Gruppen 2 und 3 erfolgt zusammen mit den entsprechenden Datentypen. In diesem Unterpunkt werden die arithmetischen Funktionen besprochen. Es stehen zur Verfügung:

| | |
|---|---|
| SQR(X) | Quadrat |
| SQRT(X) | Quadratwurzel |
| ABS(X) | Absolutwert |
| SIN(X) | Winkelfunktion |
| COS(X) | Winkelfunktion |
| ARCTAN(X) | Winkelfunktion |
| EXP(X) | e-Funktion |
| LN(X) | natürlicher Logarithmus |

Der aktuelle Parameter kann ein beliebiger Ausdruck vom Typ REAL oder INTEGER sein. Das Resultat ist stets REAL. Eine Ausnahme bildet ABS. Hier hat das Resultat stets den Typ des aktuellen Parameters.
Zur Umwandlung von REAL in INTEGER gibt es die Funktionen

| | |
|---|---|
| TRUNC(X) | ganzer Teil |
| ROUND(X) | regelgerechte Rundung. |

Die nachfolgenden Standardfunktionen sind auf einfache Typen mit Ausnahme von REAL anwendbar:

| | |
|---|---|
| ODD(A) | Test auf ungerade |
| ORD(A) | interne Kodierung von A |
| SUCC(A) | Nachfolger von A |
| PRED(A) | Vorgänger von A |
| CHR(I) | Konstante vom Typ CHAR mit interner Kodierung I |

Die Funktion CHR kann vorteilhaft zur Konstruktion von CHAR-Konstanten für nicht darstellbare Zeichen des ASCII-Kodes genutzt werden. Zum Beispiel liefert CHR(27) den Kode für Escape.
Die Wirkung der Standardfunktionen ORD, SUCC und PRED wird durch die Anwendung auf den Aufzählungstyp

```
TYPE T = (Mo,Di,Mi,Don,Fr,Sa,So);
VAR X : T;
```

gezeigt:

```
ORD(Mo) liefert: 0
ORD(Di) liefert: 1
.
.
.
ORD(So) liefert: 6
```

Die Anwendung von Nachfolger- und Vorgängerfunktionen liefern:

```
SUCC(Mo) liefert: Di
SUCC(Di) liefert: Mi
.
.
.
SUCC(Sa) liefert: So

PRED(So) liefert: Sa
PRED(Sa) liefert: Fr
.
.
.
PRED(Di) liefert: Mi
```

Nicht definiert sind die Funktionswerte SUCC(So) und PRED(Mo).
Darüber hinaus stellen viele PASCAL-Systeme weitere Standardprozeduren und -funktionen bereit, mit denen Eigenschaften der Gerätetechnik oder des zugrunde liegenden Betriebssystems auf das Niveau der Sprache gehoben werden. In TURBO-PASCAL gibt es davon mehr als 50. Eine Auswahl wird hier angegeben:

| | |
|---|---|
| ADDR(X) | Adresse der Variablen X |
| SIZEOF(X) | Speicherplatzbedarf von X in Byte |
| LO(X) | Niederwertiges Byte von X |
| HI(X) | Höherwertiges Byte von X |
| SWAP(X) | Vertauschen der Bytes von X |
| MOVE(VON,ZU,ANZ) | Blocktransport von "ANZ"-Bytes |
| BIOS(I,J) | BIOS-Ruf I mit Parameter J |
| BDOS(I,J) | BDOS-Ruf I mit Parameter J |
| HALT | Abbruch der Programmabarbeitung |

## 6. Datentyp File, Ein- und Ausgabeorganisation

### 6.1. Einführung

Der Datentyp File bildet in PASCAL die Grundlage für die Ein- und Ausgabe von Daten von bzw. zu den peripheren Speichermedien.
Ein Datenobjekt vom Filetyp ist eine Folge von Komponenten gleichen Typs. Der Typ einer Komponente ist beliebig. Der Zugriff zu den Komponenten ist nur sequentiell möglich.
Die Brücke zu den peripheren Geräten wird geschlagen, indem Variablen vom Filetyp als Files im Sinne eines Betriebssystems angelegt werden.

### 6.2. Deklaration und Zugriff

Die Deklaration von Filetypen ist eine Alternative im Syntaxdiagramm "typ" (Bild 3.1). Sie wird im Bild 6.1 gezeigt. Die reservierten Bezeichner FILE und OF leiten die Deklaration ein. Anschließend wird der Komponententyp angegeben. Jeder Filevariablen wird eine Puffervariable zugeordnet. Sie wird gebildet durch Anfügen des Zeichens "^" an den Bezeichner einer Filevariablen.

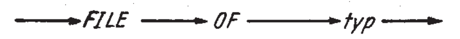



*Bild 6.1 Syntaxdiagramm „filetyp"*

Über die Puffervariable kann auf die sogenannte aktuelle Komponente eines Files zugegriffen werden. Sie läßt sich wie eine Variable verwenden.
Es folgen einige Beispiele:

a) Komponententyp INTEGER:

```
VAR F: FILE OF INTEGER;
    I: INTEGER;
.
F^:=1; I:=2*F^+1; ...
```

b) Komponententyp REAL

```
VAR F: FILE OF REAL;
    X: REAL;
X:=F^*2.5; F^:=0.5; ...
```

c) Komponente vom Aufzählungstyp

```
TYPE T=(Mo,Di,Mi,Don,Fr,Sa,So);
VAR F: FILE OF T;
    Z:T;
.
.
.
F^:=Di; Z:=F^; ...
```

F^ ist die Puffervariable. Durch die Zugriffe wird die aktuelle Komponente nicht verstellt. Dazu gibt es Standardprozeduren, die im nächsten Unterpunkt besprochen werden.

*6.3. Standardprozeduren und -funktionen für den Datentyp File*

Die Ein- und Ausgabe erfordert einige komplexe Leistungen. Sie werden in Form von Standardprozeduren und -funktionen bereitgestellt. In der nachfolgenden Aufstellung bezeichnet f eine Variable vom Filetyp:

| | |
|---|---|
| RESET(f) REWRITE(f) | Vorbereiten des Files für Lesen bzw. Schreiben. Die Puffervariable wird auf die erste Komponente positioniert. |
| GET(f) PUT(f) | Weiterrücken der Puffervariablen auf die nächste Komponente des Eingabe- bzw. Ausgabefiles |
| EOF(f) | Funktion, liefert den Wert TRUE, falls Fileende erreicht ist. |

Bei Eingabefiles zeigt EOF(f)=TRUE an, daß die Puffervariable auf Fileende positioniert wurde und keine weiteren Komponenten vorhanden sind. Der Zugriff über die Puffervariable f^ liefert einen unbestimmten Wert.
Für Ausgabefiles zeigt EOF(f)=TRUE an, daß die Puffervariable auf das Fileende positioniert ist und die nächste Komponente angefügt werden kann.
Mit diesen fünf Prozeduren kann die gesamte Ein- und Ausgabe durchgeführt werden. Später werden weitere Prozeduren besprochen, die die Arbeit erleichtern, aber keine prinzipiell neuen Leistungen erbringen.
Es gibt zwei Grundaufgaben bei der Ein- und Ausgabe: das Schreiben und das Lesen eines Files. Die programmtechnische Lösung wird jetzt gezeigt. Sie ist unabhängig vom Typ der Komponenten. Er wird deshalb auch offen gelassen:





```
PROGRAM Schreiben;
TYPE Komp = ...;
VAR F:FILE OF Komp;
    X:Komp;
    I:INTEGER;
BEGIN
  REWRITE(F);
  FOR I:=1 TO 100 DO BEGIN
    F^:=X; PUT(F)
  END
END.
```

```
PROGRAM Lesen;
TYPE Komp = ...;
VAR F:FILE OF Komp;
    X:Komp;
BEGIN
  WHILE NOT EOF(F) DO BEGIN
  X:=F^; GET(F)
  END
END.
```

Natürlich hätte das Lesen auch innerhalb einer FOR-Anweisung erfolgen können. Die gezeigte Variante hat aber den Vorteil, daß sie auch funktioniert, wenn die Anzahl der Komponenten unbekannt ist.
Beim Lesen findet keine Überprüfung des Komponententyps statt. Weiterhin erlaubt Standard-PASCAL für ein File nur Lese- oder Schreibzugriffe. Das Verändern eines Files muß durch Kopieren gelöst werden.
Durch zwei zusätzliche Standardprozeduren wird die Arbeit mit Filevariablen weiter vereinfacht:

READ(f,v) entspricht v:=f^; GET(f)
WRITE(f,a) entspricht f^:=a; PUT(f).

Hier stehen v für eine Variable und a für einen Ausdruck vom Komponententyp der Filevariablen f.
Als weitere Vereinfachung sind Listen von Variablen bzw. Ausdrücken erlaubt.:

READ(f,$v_1$,$v_2$, ...)
WRITE(f,$a_1$,$a_2$, ...).

Unglücklicherweise weist TURBO-PASCAL einige Abweichungen auf:

1. Die Prozeduren PUT und GET gibt es nicht.
2. Auf die Puffervariable kann nicht Bezug genommen werden.
3. Mit der Standardprozedur ASSIGN(f,a) muß der Filevariablen f eine Datei zugeordnet werden. Der Wert des Ausdrucks a muß eine Zeichenkette sein.
4. Nach Abschluß der Arbeit ist insbesondere für Ausgabefiles durch Aufruf der Standardprozedur CLOSE(f) das Leeren der Puffer zu organisieren.

Die für TURBO-PASCAL modifizierte Lösung der Grundaufgaben wird jetzt gezeigt. Zuerst wieder das Schreiben auf die Datei A:TEST-.DAT :

```
...
BEGIN
  ASSIGN(F,'A:TEST.DAT');
  REWRITE(F);
  FOR I:=1 TO 100 DO WRITE(F,X);
  CLOSE(F)
END.
```

Und das Lesen:

```
...
BEGIN
  ASSIGN(F,'A:TEST.DAT');
  RESET(F);
  WHILE NOT EOF(F) DO READ(F,X)
END.
```

Der Typ der Komponenten kann auch hier beliebig sein.
Treten Fehler während der Ein- oder Ausgabe auf, wird in der Regel die Abarbeitung des PASCAL-Programms abgebröchen und eine Fehlermeldung ausgegeben.
TURBO-PASCAL ermöglicht die Behandlung von Ein- und Ausgabefehlern auf dem Niveau von PASCAL. Durch Aufruf der Standardfunktion IORESULT wird der Status der zuletzt durchgeführten Ein- oder Ausgabe zurückgegeben. Eine Auswahl der wichtigsten Werte folgt:

| | |
|---|---|
| 000 | Kein Fehler |
| 001 | File exisitiert nicht |
| 002 | File nicht für Eingabe eröffnet |
| 003 | File nicht für Ausgabe eröffnet |
| 004 | File nicht eröffnet |
| 091 | Positionieren über Fileende |
| 240 | Disk-Schreibfehler |
| 241 | Directory voll |
| 242 | File zu groß |
| 255 | File verloren |

Damit die Ausführung nutzereigener Fehlermaßnahmen möglich ist, müssen die Standardmaßnahmen des PASCAL-Systems unterdrückt werden.
Das ist mit der Compileroption ○I− möglich. Die Standardmaßnahmen werden durch ○I+ wieder aktiviert.

*6.5. Textfiles*

In nahezu allen EDV-Projekten spielt die Verarbeitung von Erfassungsbelegen, Drucklisten und Bildschirmmenüs eine wichtige Rolle. In PASCAL können diese Datenstrukturen durch

TYPE TEXT = FILE OF CHAR

beschrieben werden.
Das Problem besteht nun darin, daß Texte in Zeilen und Seiten strukturiert sind und daß die Repräsentation dieser Zeilen und Seiten vom jeweiligen Rechner und Betriebssystem abhängen. Aus diesem Grunde wird die Zeilen- und Seitenstruktur durch Standardprozeduren auf das Niveau von PASCAL gehoben:

| | |
|---|---|
| READLN(f) | Positionierung der Puffervariablen auf den nächsten Zeilenanfang |
| WRITELN(f) | Zeilenabschluß im Ausgabefile |
| EOLN(f) | Funktion, liefert den Wert TRUE, falls in einem Eingabefile das Zeilenende erreicht ist |
| PAGE(f) | Beginn einer neuen Seite im Ausgabefile. |

Auch hier gibt es wieder die Vereinfachungen:

READLN(f,v1,v2, ...)
für READ(f,v1,v2, ...); READLN(f)
WRITELN(f,a1,a2, ...)
für WRITE(f,a1,a2, ...); WRITELN(f)

Die Grundaufgabe bei der Arbeit mit Textfiles besteht im Erkennen und Erzeugen der Zeilenstruktur. Die programmtechnische Lösung wird im Bild 6.2 gezeigt. Den Kern des Programms bilden zwei ineinandergeschachtelte While-Anweisungen. Mit der inneren wird eine Zeile von INFILE auf OUTFILE kopiert. Die While-Anweisung wird am Zeilenende wegen EOLN(INFILE)=TRUE verlassen. Durch Aufruf von READLN(INFILE) muß nun auf die nächste Zeile im Eingabefile vorgesetzt werden. Mit WRITELN(OUTFILE) wird die Zeile im Ausgabefile abgeschlossen. Die äußere While-Schleife sorgt nun dafür, daß das Zeilenkopieren solange durchgeführt wird, bis das Fileende des Eingabefiles erreicht ist.

```
PROGRAM COPY;
VAR C: CHAR;
    INFILE,OUTFILE: TEXT;
BEGIN
  ASSIGN(INFILE,...); RESET(INFILE);
  ASSIGN(OUTFILE,...); REWRITE(OUTFILE);
  WHILE NOT EOF(INFILE) DO BEGIN
    WHILE NOT EOLN(INFILE) DO BEGIN
      READ(INFILE,C); WRITE(OUTFILE,C)
    END;
    READLN(INFILE); WRITELN(OUTFILE)
  END;
  CLOSE(OUTFILE)
END.
```

*Bild 6.2 Kopierprogramm für Textfiles*

Die PASCAL-Systeme vereinfachen die Arbeit mit Textfiles weiter. Für die Datentypen INTEGER, REAL und BOOLEAN werden Konvertierungen durchgeführt. Bei WRITE sind als aktuelle Parameter Zeichenketten zugelassen. So ist folgendes Programmstück korrekt:

```
VAR INFILE,OUTFILE: TEXT;
    I: INTEGER; B: BOOLEAN;
    X: REAL;
    ...
READ(INFILE,I,B,X);
WRITE(OUTFILE,3*I,B,'X:= ',X);
```

Im Eingabefile wirken alle Nichtziffern als Begrenzer für Zahlen. Beliebige Bezeichner, die mit den Buchstaben F oder T beginnen, werden als FALSE oder TRUE interpretiert.
Bei WRITE können die Ausdrücke mit Angaben zur Formatgestaltung versehen werden. Sie haben die Form:

ausdruck:m oder
ausdruck:m:n.

*(wird fortgesetzt)*

# PASCAL

(Teil 4)

Dr. Claus Kofer
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

Die zur Darstellung insgesamt zu verwendenden Stellen werden durch m festgelegt, die nach dem Dezimalpunkt durch n. Beispiele mit den entsprechenden Ausgaben sind:

**WRITE(OUTFILE,123:8)** liefert:
**123**
**WRITE(OUTFILE,1.5:8:3)** liefert:
**1.500**
**WRITE(OUTFILE,TRUE:8)** liefert:
**TRUE**
**WRITE(OUTFILE,'A':8)** liefert:
**A**

### 6.5. Standardfiles

PASCAL stellt zwei vordeklarierte Files bereit:

**VAR INPUT,OUTPUT: FILE OF CHAR;**

Falls sie benutzt werden sollen, sind sie in der Environmentliste anzugeben.
Ihre Zuordnung zu Files im Sinne des Betriebssystems ist bei den PASCAL-Systemen unterschiedlich. Bei TURBO-PASCAL korrespondieren INPUT und OUTPUT mit dem Gerät CON:. Abweichend vom Standard brauchen sie nicht in der Environmentliste angegeben werden. Ebenso ist die Ausführung von RESET oder REWRITE verboten.
Die Arbeit mit dem Bildschirm wird bei TURBO-PASCAL durch die Standardprozedur GOTOXY(z,s) und CLRSCR zur Cursorpositionierung bzw. zum Bildschirmlöschen unterstützt.

### 6.6. Direktzugriffsfiles

Direktzugriffsfiles gibt es in Standard-PASCAL nicht. Viele Systeme stellen sie jedoch aufgrund ihrer Wichtigkeit bereit. Die gewählte Lösung ist sehr einfach: Die Komponenten werden fortlaufend numeriert. Der Zugriff auf sie erfolgt über die Komponentennummer. In verschiedenen Systemen wird die Komponentennummer n als zusätzlicher Parameter der Prozeduren GET und PUT angegeben:

**GET**(f,n) und **PUT**(f,n).

TURBO-PASCAL verwendet einen anderen Mechanismus: Vor dem Lesen oder Schreiben wird die Puffervariable mit der Standardprozedur

**SEEK**(f,n)

auf die Komponente n positioniert. Die Zählung der Komponenten beginnt bei Null.
Weiter gibt es in TURBO-PASCAL die Standardfunktionen zur Arbeit mit Direktzugriffsfiles, die folgende INTEGER-Resultate liefern:

**FILEPOS**(f) Aktuelle Position der Puffervariablen im File f.
**FILESIZE**(f) Gesamtanzahl der Komponenten des Files f.

## 7. Datentyp Array

### 7.1. Einführung

Der Datentyp Array wird durch eine feste Anzahl von Datenelementen gleichen Typs gebildet. Die einzelnen Datenelemente heißen Komponenten. Ihre Anzahl muß zur Übersetzungszeit feststehen.
Für den Typ der Komponenten gibt es keine Einschränkungen. Insbesondere kann auch er wieder ein Array sein.
Der Zugriff auf die Komponenten erfolgt mit Hilfe von Indizes. Indizes können berechnet werden.

### 7.2. Syntax

Die Arraydeklaration ist eine Alternative des Syntaxprogramms "typ". Sie wird im Bild 7.1 gezeigt.
Sie beginnt mit dem reservierten Bezeichner ARRAY. Dann folgt eine in eckige Klammern

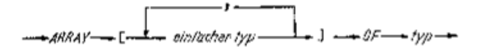



*Bild 7.1 Syntaxdiagramm "arraytyp"*

eingeschlossene Liste von einfachen Typen. REAL und INTEGER sind hier nicht zugelassen. Diese Liste gibt die Indizes an. Auf den reservierten Bezeichner OF folgt die Angabe des Komponententyps.
Die Syntax für den Zugriff zu den Komponenten eines Arrays zeigt die obere Alternative in Bild 5.13. Die Ausdrücke liefern den Wert der entsprechenden Indizes. Der Typ der Ausdrücke muß mit dem der Deklaration übereinstimmen.
Der nachfolgende Arraytyp besteht aus Komponenten vom Typ REAL. Der Indextyp ist Teilbereich des Basistyps INTEGER.

```
TYPE Index = 1...900;
     Vektor = ARRAY[Index]OF REAL;
```

Mit diesem Arraytyp können die Variablen

**VAR X,Y,Z** : Vektor;

vereinbart werden.
Zugriffe auf die Komponenten sind dann wie folgt möglich:

**X[1] , X[3*I+K] , Y[I DIV 2], Z[SQR(I)−1]**

Syntaktisch korrekt ist auch die Form

**VAR X,Y,Z:ARRAY [1..100] OF REAL;**

Hier ist jedoch der Typ der Variablen X,Y,Z anonym und kann zu keiner weiteren Deklaration genutzt werden.
Ein weiteres Beispiel für ein Array ist:

```
TYPE Tag= (Mo,Di,Mi,Don,Fr,Sa,So);
     Arbeitszeit=
     ARRAY [Tag] OF REAL;
VAR  MeiersZeit : Arbeitszeit;
```

Der Komponententyp ist REAL. Der Indextyp ist ein Aufzählungstyp. Zugriffe auf die Komponenten der Variablen MeiersZeit sind

MeiersZeit[Mo]
MeiersZeit[succ(Mo)]

Zur Deklaration von Arrays mit mehreren Indizes gibt es zwei Interpretationen des Syntaxdiagramms, s. Bild 7.1:

```
ARRAY[index1] OF
  ARRAY [index2] OF..typ
und
  ARRAY[index1,index2,...] OF typ
```

Beide Formen sind korrekt und unterscheiden sich inhaltlich nicht.
Für den Zugriff zu den Komponenten mehrfach indizierter Arrays gibt es nach Syntaxdiagramm 5.13 ebenfalls zwei Formen, die sich inhaltlich nicht unterscheiden:

bezeichner [ausdruck1][ausdruck2]...
und
bezeichner [ausdruck1,ausdruck2,...].

Wesentlich ist, daß die Anzahl und der Typ der Ausdrucke mit denen bei der Deklaration übereinstimmen muß.
Eine quadratische Matrix läßt sich wie folgt deklarieren:

```
TYPE Index = 1..Max;
     Matrix = ARRAY [Index,Index]
                    OF REAL;
VAR  A : Matrix;
```

Der Zugriff zu den Komponenten hat dann die Form

**A[1,3],A[I,J],A[3-I,I DIV 2]**

Die Typen der Arrayindizes können natürlich auch alle voneinander verschieden sein:

```
TYPE Bsp =
     ARRAY[1..3,BOOLEAN,CHAR]
           OF INTEGER;
VAR  X : Bsp;
```

Komponenten von X werden z. B. durch

**X[2,TRUE,'A']**,
**X[I,ODD(J),CHR(K)]**. usw.

ausgewählt.
Die PASCAL-Systeme sichern, daß beim Zugriff auf eine Komponente die Indizes innerhalb ihrer Wertebereiche liegen. Das verlangsamt die Abarbeitung des Programms

und führt zu größerem Programmkode. Deshalb können diese Kontrollen meist ein- und ausgeschaltet werden. Bei TURBO-PASCAL erfolgt das durch die Compileroption ○R+ bzw. ○R–.
Standard-PASCAL sieht das Packen von Arrays vor. Der Compiler wird durch Einfügen der reservierten Bezeichnung PACKED vor ARRAY angewiesen, für die Komponenten des Array eine speicherplatzsparende interne Darstellung zu wählen. Das erfolgt z. B. dadurch, daß für Teilbereichstypen nur noch so viele Bits vorgesehen werden, wie zur Darstellung ihres Wertbereiches tatsächlich notwendig sind.
Der Zugriff auf gepackte Datenstrukturen ist deshalb langsamer.
Gepackte Arrays mit dem Komponententyp CHAR sind mit Zeichenketten verträglich.
TURBO-PASCAL führt kein Packen durch. Es überliest PACKED, aber Arrays mit dem Komponententyp CHAR sind mit Zeichenketten verträglich.

### 7.3. Operationen

PASCAL gestattet für Arrays gleichen Typs die Zuweisung und die Ausführung von Vergleichsoperationen.
Mit den Komponenten können die für ihren Typ zulässigen Operationen ausgeführt werden.
Das erste Beispiel zeigt die Zuweisung von Arrays:

```
TYPE Index = ...;
     Mat = ARRAY[Index,Index]
              OF REAL
VAR  A,B : Mat;
...
A := B;
...
```

Wichtig ist, daß die Variablen den gleichen Typ haben. Aus der Sicht des Compilers heißt das: Deklaration durch den gleichen Typbezeichner oder den gleichen impliziten Typ. Strukturelle Gleichheit wie z. B.

```
VAR A:ARRAY[Index,Index]
          OF REAL;
    B:ARRAY[Index,Index]
          OF REAL;
```

erkennt der Compiler nicht.
Vergleiche können häufig vorteilhaft auf Arrays mit dem Komponententyp CHAR angewendet werden. Das folgende Programmstück zeigt ein Beispiel, bei dem in einem Array von Namen ein Schlüssel gesucht wird. Namen und Schlüssel sind selbst Arrays:

```
TYPE Alpha = ARRAY[1..10]
               OF CHAR;
VAR  key:Alpha;
     Namen:ARRAY[1..100]
               OF Alpha;
     I:INTEGER;
...
I:=0;
REPEAT
I:=I+1;
UNTIL Namen[I] = Key;
...
```

Den Arrays mit dem Komponententyp CHAR können Zeichenketten zugewiesen werden. Die Länge der Zeichenkette muß dabei mit der Anzahl der Komponenten des Arrays übereinstimmen.

### 7.4. Programmbeispiel

Der Datentyp Array ist die programmiersprachliche Entsprechung der aus der Mathematik bekannten Vektoren und Matrizen. So bieten sich Programmbeispiele aus der linearen Algebra geradezu an. Trotzdem soll hier eine andere Anwendung gezeigt werden: In einem Textfile ist die Häufigkeit des Auftretens der Buchstaben A ... Z zu ermitteln. Die Häufigkeit dieser Zeichen wird in dem Array H mit folgender Deklaration gezählt:

```
H:ARRAY[CHAR]OF REAL
```

Das zeichenweise Lesen eines Textfiles ist bereits aus Abschnitt 6.5. bekannt. So erhält man mit wenigen Erweiterungen die im Bild 7.2 gezeigte programmtechnische Lösung.

```
PROGRAM Zaehl;
VAR C: CHAR;
    H: ARRAY[CHAR] OF REAL;
    F: TEXT;
BEGIN
  FOR C:='A' TO 'Z' DO H[C]:=0.0;
  ASSIGN(F,...); RESET(F);
  WHILE NOT EOF(F) DO BEGIN
   WHILE NOT EOLN(F) DO BEGIN
    READ(F,C); H[C]:=H[C]+1
   END;
   READLN(F)
  END;
  FOR C:='A' TO 'Z' DO
    WRITELN(C,H[C])
END.
```

*Bild 7.2 Programm zum Zählen der Zeichen in einem Textfile*

## 8. Datentyp Record

### 8.1. Einführung

Der Datentyp Record wird durch eine feste Anzahl von Datenelementen gebildet, deren Typ unterschiedlich sein kann. Die einzelnen Datenelemente eines Records heißen Felder. Zu ihnen kann über sogenannte Feldbezeichner zugegriffen werden. Feldbezeichner können nicht berechnet werden. Ihr Name ist fest im Programmtext verankert.
Für den Typ der Recordfelder gibt es keine Einschränkungen. Insbesondere können sie auch selbst wieder Records sein.

### 8.2. Syntax

Die Syntax des Recordtyps wird in den Bildern 8.1 bis 8.3 gezeigt. Die reservierten Bezeichner RECORD und END rahmen die Feldliste ein. Ihr Aufbau wird im Bild 8.2 gezeigt. Sie besteht aus einzelnen Feldern, die durch einen Feldbezeichner und eine Typenangabe gebildet werden. Falls aufeinanderfolgende Recordfelder den gleichen Typ haben, können ihre Bezeichner zu einer durch

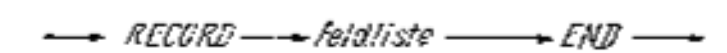



*Bild 8.1 Syntaxdiagramm "recordtyp"*

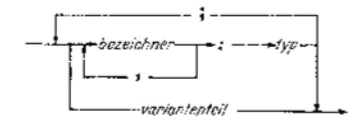



*Bild 8.2 Syntaxdiagramm "feldliste"*

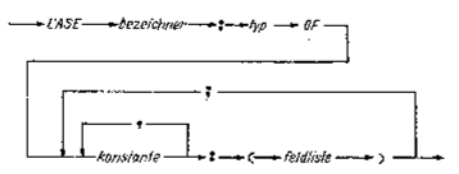



*Bild 8.3 Syntaxdiagramm "variantenteil"*

Komma getrennten Liste zusammengefaßt werden.
Die Feldliste kann durch einen Variantenteil abgeschlossen werden. Seine Behandlung erfolgt im Abschnitt 8.5.
Die Feldbezeichner haben nur innerhalb des Records Bedeutung. Sie überdecken bereits deklarierte Namen von Konstanten, Typen oder anderen Variablen nicht. Insbesondere können auch in verschiedenen Recordtypen gleiche Feldbezeichner verwendet werden.
Der Zugriff zu den Feldern eines Records ist bereits im Syntaxdiagramm in Bild 5.13 als untere Alternative dargestellt. Die einfachste Form sind zwei aufeinanderfolgende Bezeichner, die durch einen Punkt getrennt werden. Der erste Punkt muß der Name einer Recordvariablen, der zweite ein Feldbezeichner sein. Der Punkt hat die Aufgabe eines Selektors. Er wählt aus dem Record ein Feld aus. Ist dieses Feld selbst ein Record, kann ein weiterer Feldbezeichner angefügt werden. Das Syntaxdiagramm in Bild 5.13 erlaubt dies.
Es folgen einige Beispiele: Der Recordtyp

```
TYPE Complex = RECORD
       R : REAL;
       I : REAL
     END;
```

besteht aus zwei Feldern vom Typ REAL.
Die Feldbezeichner sind R und I. Da beide Felder den gleichen Typ haben, läßt sich auch abkürzend schreiben:

```
TYPE Complex = RECORD
       R,I : REAL
     END;
```

Vor dem abschließenden END ist lt. Syntax kein Semikolon erlaubt. Mit dem nachfolgenden Record wird ein Datentyp zur Beschreibung eines Datums eingeführt:

```
TYPE Datum = RECORD
       Jahr: 0..3000;
       Monat: (Jan,Feb,Mar,Apr,Mai,
       Jun,Jul,Aug,Sep,Okt,Nov,Dez);
       Tag: 1..31
     END
```

Der Record besteht aus drei Feldern mit den Bezeichnern Jahr, Monat und Tag. Die entsprechenden Datentypen dieser Felder sind Teilbereich von INTEGER, Aufzählungstyp und wieder Teilbereich von INTEGER.
Eine Komponente in einem File von Studenten könnte folgender Recordtyp Person sein:

```
Type  Alpha = ARRAY[1..12]
                     OF CHAR;
      Person = RECORD
        Name: Alpha;
        Vorname: Alpha;
        Semgr: 1..15
      END;
```

Zu ineinander geschachtelten Records kommt man sofort, wenn der Typ Person durch Angaben zum Datum der Geburt und der Immatrikulation erweitert wird.

```
TYPE  Alpha = ...;
      Datum = ...;
      Person = RECORD;
        Name: Alpha;
        Vorname: Alpha;
        GebDat: Datum;
        ImatDat: Datum;
        Semgr: 1..15
      END;
```

Der Zugriff auf die Felder der angegebenen Records wird am Beispiel der Variablen

```
VAR Y,Z: Complex;
    Jetzt: Datum;
    Student: Person;
```

gezeigt:
**Y.R, Y.I, Z.R, Z.I**
Jetzt.Jahr, Jetzt.Monat, Jetzt.Tag
Student.Name, Student.GebDat.Jahr, Student.GebDat.Monat, usw.

### 8.3. Operationen mit Records

PASCAL gestattet für Records gleichen Typs die Zuweisung und die Vergleichsoperationen. Es wird noch einmal daran erinnert, daß die Compiler keine strukturelle Gleichheit erkennen. Siehe dazu auch Punkt 7.3.
Auf die Recordfelder können die für ihren Typ zulässigen Operationen angewendet werden.
Es folgen einige Beispiele auf der Grundlage der im vorigen Punkt eingeführten Recordtypen Complex, Datum und Person mit folgenden Variablen:

```
VAR  Y,Z:Complex;
     Jetzt: Datum;
     Student: Person;
     Jahrgang88:
       ARRAY[1..300]OF PERSON
     ...
     Y:= Z; Y.R:=1.0;
     Y.R:=Y.R+Z.R; Y.I:=Y.I+Z.I;
```

Die letzte Zeile ist ganz offensichtlich die komplexe Addition, aber Y:=Y+Z; ist nicht erlaubt, da die Addition für Records nicht erklärt ist.

```
Jetzt.Jahr:= 1987;
Jetzt.Monat:= Jun;
Jetzt.Tag:= 11;
```

Das Datum, an dem diese Zeilen geschrieben wurden.

```
Student.Name:= 'Zimmermann';
Student.GebDat.Jahr:= 1968;
Student:= Jahrgang88[I]
Jahrgang88[I].Semgr:= 1;
```

Im letzten Beispiel wird zunächst eine Komponente im ARRAY Jahrgang88 ausgewählt. Sie ist ein Record, also kann mit dem Selektor ein Feld ausgewählt werden.

### 8.4. With-Anweisung

Die With-Anweisung gehört zu den Anweisungen, beeinflußt jedoch den Prozeßablauf nicht. Sie dient der Reduzierung von Schreibarbeit bei Bezugnahmen auf Recordfelder.
Das Bild 8.4 zeigt die Syntax der With-Anweisung. Ihre Wirkung besteht ausschließlich darin, daß sie für die zu ihr gehörende Anweisung die Feldbezeichnung der zwischen WITH und DO aufgeführten Recordvariablen bekannt macht.

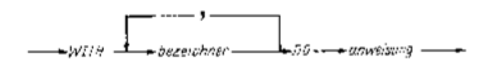



*Bild 8.4 Syntaxdiagramm "with-anweisung"*

Bei den gezeigten Beispielen darf unter Verwendung der With-Anweisung geschrieben werden:

```
WITH Y DO R:= 1.0
WITH Jetzt DO Jahr:= 1987
WITH Student DO
     Name:= 'Zimmermann'
```

Auch hier darf die Verbundanweisung benutzt werden, wenn eine einzige Anweisung nicht ausreicht:

```
WITH Jetzt DO BEGIN
  Jahr:= 1987;
  Monat:= Jun,
  Tag:= 11
END
```

Die allgemeine Form der Anweisung

WITH r1,r2,...rn DO anweisung

entspricht

```
WITH r1 DO
  WITH r2 DO
  ...
  WITH rn DO anweisung
```

Innerhalb der With-Anweisung können durch die Feldbezeichner eines Records andere Bezeichner überdeckt werden. Das folgende Programmstück zeigt dies:

```
TYPE Complex = ...;
VAR  I:BOOLEAN;
     X:Complex;
WITH X DO I:=1.0;
```

Der Feldbezeichner I überdeckt die Variable I vom Typ BOOLEAN. Auf sie kann innerhalb der With-Anweisung nicht Bezug genommen werden. Zur Beschleunigung des Zugriffs auf die Felder eines mit der With-Anweisung ausgewählten Records wird die Anfangsadresse des Records meist gesondert abgespeichert. Der dafür erforderliche Platz ist um so größer, je mehr With-Anweisungen verschachtelt sind. TURBO-PASCAL plant standardmäßig eine Tiefe von vier und reserviert im Datenbereich einer jeden Prozedur dafür Platz, unabhängig davon, ob diese Tiefe tatsächlich erreicht wird. Durch die OWN-Option kann der Standard auf eine Tiefe n zwischen 0 und 9 geändert werden.

### 8.5. Variantenrecords

Zur Motivation von Variantenrecords wird noch einmal der Record Person in seiner ersten Form aufgegriffen. Er enthält zur Kennzeichnung eines Studenten die drei Felder Name, Vorname und SemGr. Sollen mit dem Record Person auch Daten von anderen Beschäftigten erfaßt werden, hat das Feld SemGr keinen Sinn. Dafür werden aber andere Informationen benötigt, z. B. Wissenschaftsbereich, Gebäude, Zimmernummer und Telefon. Der Record Person sollte zwei verschiedene Bestandteile haben: einen feststehenden, mit den Feldern Name und Vorname, und einen, der entweder durch die Felder Seminargruppe oder Wohnheim gebildet wird, falls es sich um einen Studenten handelt oder durch Wissenschaftsbereich, Gebäude, Zimmernummer und Telefon, falls es ein Mitarbeiter ist. Eben diese Möglichkeit eröffnen Variantenrecords. Die Syntax zeigt Bild 8.3. Die Aufgabe der einzelnen Bestandteile wird am folgenden Beispiel gezeigt:

```
TYPE Alpha = ...;
     PersKat = (ST,MA);
     Heime = (WH1,WH2,WH3);
     StrEinh = (WB1,WB2,WB3,WB4);
     Bauwerk = (BW1,BW2);
     Person = RECORD
     Name, Vorname: Alpha;
     CASE PArt: PersKat OF
     ST:(SemGr: 1..15;
          Wohnh: Heime);
     MA:    (WB:StrEinh;
            Geb: Bauwerk;
            Zi: 1..500;
            Tel: 100..999)
END;
```

Der reservierte Bezeichner CASE leitet die Varianten ein. Das Feld PArt ist das sogenannte Anzeigefeld. Es folgen nun die Varianten des Records. Dabei kann für jeden Wert des Anzeigefeldes eine Struktur angegeben werden. Im Beispiel sind das zwei.
Bei der Arbeit mit Variantenrecords ist der Programmierer für den korrekten Zugriff zu den Feldern des Variantenteils selbst verantwortlich. In der Regel wird dazu das Anzeigefeld zu Hilfe genommen.

Das folgende Programmstück zeigt dies:

```
VAR P: Person;
...
WITH P DO BEGIN
  IF PArt = St THEN
    WRITE ('SemGruppe=',SemGr)
  ELSE
    WRITE('Tel-Nr.=',Tel)
  END;
  ...
```

Falls die Unterscheidung der Varianten nicht notwendig ist, kann die Abspeicherung des Anzeigefeldes entfallen. Es wird bei der Deklaration nur sein Datentyp angegeben.
Variantenrecords lassen sich verwenden, um die Typenkontrollen des Compilers zu unterlaufen. Das folgende Beispiel zeigt den Zugriff zum nieder- und höherwertigen Byte eines Datenelementes vom Typ INTEGER:

```
TYPE Trick = RECORD
        CASE BOOLEAN OF
        FALSE:(Int:INTEGER);
        TRUE:(BY:ARRAY[0..1]
                            OF CHAR)
     END;
VAR I: Trick;
     High,Low:INTEGER;
...
I.INT:= ...;
Low:= ORD(I.BY[0]);
High:= ORD(I.By[1]);
...
```

Im Beispiel ist für das Anzeigefeld ein Typ notwendig, dessen Wertebereich aus zwei Werten besteht. Welcher das ist, hat keine Bedeutung. Hier wurde BOOLEAN gewählt, da er als Standardtyp bereits bekannt ist.

## 9. Datentyp Menge

### 9.1. Einführung

Eine Menge ist im mathematischen Sinne eine Zusammenfassung bestimmter, unterscheidbarer Elemente. Einschränkend wird in PASCAL gefordert, daß die Elemente alle vom selben einfachen Typ mit Ausnahme von REAL sind. So können z. B. die Mengen der geraden bzw. ungeraden ganzen Zahlen [2,4,6,8,10] bzw. [1,3,5,7,9] gebildet werden, nicht aber [1.5,TRUE,11], denn die Elemente gehören zu unterschiedlichen Datentypen. Die eckigen Klammern stellen hier den sogenannten Mengenkonstruktor dar. Die maximale Anzahl der Elemente einer Menge ist begrenzt. Zwischen den PASCAL-Systemen gibt es hier Unterschiede. Übliche Werte sind 64, 128 oder 256. TURBO-PASCAL beschränkt Mengen auf 256 Elemente. Den Wertebereich einer Variablen von Mengentyp bildet die Potenzmenge über der Basismenge. Die Potenzmenge enthält 2**n Teilmengen, wobei n die Anzahl der Elemente der Basismenge ist.

### 9.2. Syntax

Der Mengentyp ist eine Alternative von "typ" im Bild 3.1. Seine Syntax zeigt Bild 9.1. Die Deklaration von Mengen beginnt mit den reservierten Bezeichnern SET und OF. Es folgt die Angabe des Basistyps, der ein einfacher Typ (ohne REAL) sein muß.
Mengenkonstruktoren gestatten die Bildung von Mengenkonstanten. Sie können in Ausdrücken als Faktor verwendet werden. Ihre Syntax zeigt Bild 9.2. Es gibt zwei Grundformen:

[ausdruck, ausdruck, ...]
oder
[ausdruck .. ausdruck, ...]

In der ersten Form werden die Ausdrücke ausgewertet und die entsprechenden Werte des Basistyps in die Menge aufgenommen. Bei der zweiten Form werden die Ausdrücke ebenfalls ausgewertet. Die Resultate grenzen einen Bereich des Basistyps ein, aus dem alle Werte in die Menge aufgenommen werden. Es sind Kombinationen erlaubt, z. B.

[ausdruck, ausdruck .. ausdruck,ausdruck]

→ SET → OF → einfacher typ →

*Bild 9.1 Syntaxdiagramm "mengentyp"*

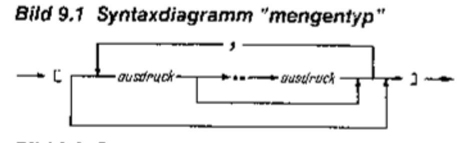



*Bild 9.2 Syntaxdiagramm "mengenkonstruktor"*

Es folgen Beispiele:

**TYPE T=SET OF (A,B,C);**

Der Wertebereich des Mengentyps T besteht aus

- den einelementigen Teilmengen **[A],[B],[C]**
- den zweielementigen Teilmengen **[A,B],[A,C],[B,C]**
- der Grundmenge **[A,B,C]**

und der Leeren Menge [ ]

Mehr als diese acht Werte (2**3=8) gibt es nicht. Mengenkonstruktoren für den Mengentyp T sind:

**[A] , [A .. C] , [B,C] , [X]**

Hier muß X eine Variable vom Typ T sein. Es wird eine Menge über einen Teilbereich von INTEGER gezeigt:

**TYPE** BitSet = **SET OF 0 .. 7;**

Der Wertebereich des Mengentyps BitSet besteht aus 256 Teilmengen (2**8=256). Erlaubte Mengenkonstruktoren sind:

**[0]**
**[1]**
**[7]**
**[0,3,5 .. 7]**
**[0 .. 7]**
**[3*I+K]**

Der Ausdruck 3*I+K muß einen Wert vom Typ INTEGER liefern und im Bereich 0 ... 7 liegen.
Mengen über dem Basistyp INTEGER können nicht deklariert werden, da der Wertebereich von INTEGER mehr als 256 Elemente umfaßt. Ein letztes Beispiel zeigt einen Mengentyp über den Basistyp CHAR:

**TYPE** CharSet = **SET OF CHAR;**

Wenn von 127 Zeichen des ASCII-Kodes ausgegangen wird, besteht der Wertbereich des Datentyps CharSet aus mehr als 1.R38 (genau 2**127) Elementen. Mengenkonstruktoren vom Typ CharSet sind z. B.

**['A' .. 'Z']**
**['0' .. '9']**
**['+','-','*','/']**
**['a' .. 'z','+','/']**
**[C]**
**['z' .. C]**

Die Variable C muß vom Typ CHAR sein.

### 9.3. Operationen mit Mengen

Für Datenobjekte vom Mengentyp gibt es folgende arithmetische Operationen:

+ Vereinigung
* Durchschnitt
− Differenz.

Weiterhin sind alle Vergleichsoperatoren anwendbar. Dabei wird getestet auf

| | |
|---|---|
| < | echte Teilmenge |
| <= | Teilmenge |
| =,<> | Gleichheit, Ungleichheit |
| >= | Obermenge |
| > | echte Obermenge. |

Durch den Operator IN kann weiterhin abgefragt werden, ob ein bestimmtes Element in einer Menge enthalten ist.
Falls M1, M2 und M3 Mengen des Typs SET OF (A,B,C) sind und die Werte M1=[A,B],M2=[B,C] und M3=[C] haben, liefern die Verknüpfungen folgende Ergebnisse:

| | | |
|---|---|---|
| **M1 + M2** | = | **[A,B,C]** |
| **M2 + M3** | = | **[B,C]** |
| **M1 * M2** | = | **[B]** |
| **M1 * M3** | = | **[ ]** |
| **M2 − M3** | = | **[B]** |
| **M1 − M2** | = | **[A]** |
| **M3 < M1** | = | **TRUE** |
| **M1 <=M1** | = | **TRUE** |
| **M1 = M2** | = | **FALSE** |
| **M1 <> M2** | = | **TRUE** |
| **M2 >=M2** | = | **TRUE** |
| **M2 > M3** | = | **TRUE** |
| **A IN M1** | = | **TRUE** |
| **C IN M1** | = | **FALSE** |

*wird fortgesetzt*

# PASCAL (Teil 5)

**Dr. Klaus Kofer**
**Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden**

### 9.4. Programmbeispiel

Mengentypen können vorteilhaft dann angewendet werden, wenn im Programm bestimmte Entscheidungen davon abhängen, ob eine Variable irgendeinen aus einer Menge von möglichen Werten hat.
Falls in dem bereits gezeigten Programm zur Ermittlung der Häufigkeit einzelner Zeichen in einem Textfile nur die Anzahl der Vokale ermittelt werden soll, könnte die Lösung ohne Verwendung von Mengen so aussehen:

```
IF(C = 'A') OR (C = 'a')  OR
  (C = 'E') OR (C = 'e')  OR
  (C = 'I') OR (C = 'i')  OR
  (C = 'O') OR (C = 'o')  OR
  (C = 'U') OR (C = 'u')  THEN...
```

Eleganter ist die Lösung mit Mengen:

```
TYPE      CharSet = SET OF CHAR;
VAR       Vokale: Char Set;
...
Vokale: = ['A', 'E', 'I', 'O', 'U',
           'a', 'e', 'i', 'o', 'u'];
...
...
IF G IN Vokale THEN...
...
```

Die Konstruktion der Menge Vokale braucht natürlich nur einmal am Beginn der Programmabarbeitung durchgeführt werden.

### 9.5. Interne Darstellung

Die interne Darstellung einer Menge ist ein Bitmuster. Die Arbeit mit Mengen entspricht der Manipulation von Bitmustern. Durch Kenntnis der internen Abläufe bei der Arbeit mit Mengen können möglicherweise weitere sinnvolle Anwendungen des Mengentyps erschlossen werden.
Durch die Deklaration

```
VAR M:SET OF(W0, W1, W2, ..., Wn);
```

plant der Compiler für die Variable M eine Anzahl von Bits. Gewöhnlich ein Vielfaches von acht, mindestens aber n+1. Die Bits werden durchnumeriert 0, 1, 2, ..., n−1, n. Nun korrespondiert jedes Bit mit einem der Werte W0, W1, ..., Wn:

```
W0 – BIT 0
W1 – Bit 1
...
Wn – Bit n
```

Gehört das Element Wi der Menge an, so hat das korrespondierende Bit i den Wert 1, sonst 0.
Aufgrund dieser Darstellung können die Mengenoperationen unmittelbar durch logische Befehle des Maschinenbefehlssystems ausgeführt werden. Bit-Setze- und Bit-Test-Befehle bewerkstelligen die Konstruktion von Mengen und den Test, ob ein Element in einer Menge enthalten ist.
Die Nützlichkeit der Mengen hängt wesentlich davon ab, wie differenziert der Compiler die Zuordnung von Speicherplatz vornimmt. Relativ unbrauchbar ist sicher eine Realisierung für Mengen, die unabhängig davon, wieviele Elemente der Wertebereich des Basistyps tatsächlich umfaßt, stets die maximale Anzahl von 64, 128 oder 256 Bits (entsprechend 8, 16 oder 32 Byte) zuordnet.
Bei TURBO-PASCAL werden nur so viele Bits geplant, wie der Basistyp Elemente hat. Dabei wird auf das volle Byte aufgerundet.

## 10. Datentyp Pointer

### 10.1. Einführung

Die Variablendeklaration veranlaßt den Compiler, Speicherplatz entsprechend dem Bedarf des jeweiligen Datentyps zu planen.
Es gibt jedoch Programme, bei denen sich erst während ihrer Arbeit – etwa in Abhängigkeit von den Eingabedaten – Menge und Typ des erforderlichen Speichers für Daten herausstellt. Hierfür ist ein Mechanismus wünschenswert, der dem Programm die „eigenverantwortliche" Abforderung vom Datenspeicher gestattet. Pointertypen ermöglichen das.

### 10.2. Syntax

Der Pointertyp ist eine Alternative im Syntaxdiagramm „typ" in Bild 3.1. Seine Deklaration zeigt Bild 10.1. Der Bezeichnung muß der Name eines Datentyps sein. Das Zeichen "^" ist zu lesen wie „Zeiger auf". Pointer verweisen also auf Datenobjekte eines bestimmten, bei der Deklaration festgelegten Typs. Der Wert einer Pointervariablen ist stets eine Hauptspeicheradresse. Der Datentyp, auf den sie verweist, hängt von ihrer Deklaration ab. Nachfolgend werden ein Pointer auf INTEGER, ein Pointer auf ein Array mit vier Komponenten vom Typ REAL und ein Pointer auf einen Record mit zwei Feldern der Typen REAL und CHAR gezeigt:

```
TYPE IntPtr = ^ INTEGER;
ArrPtr = ^ ARRAY [0...3]
                OF REAL;
RecPtr = ^ RECORD
             FA: REAL;
             FB: CHAR;
           END;
```

Obwohl die Werte aller drei Pointervariablen Hauptspeicheradressen sind, sind sie nicht miteinander verträglich. Denn sie verweisen auf Datenobjekte ganz unterschiedlichen Typs.

*Bild 10.1 Syntaxdiagramm „pointertyp"*

Zum Zugriff auf die durch Pointer referenzierten Datenobjekte wird der Pointervariablen das Zeichen "^" nachgestellt.
Falls die Variablen

```
VAR PI: IntPtr;
    PA: ArrPtr;
    RR: RecPtr;
```

deklariert sind, stellt PI^ ein Datenobjekt vom Typ INTEGER, PA^ ein Array und PR^ einen Record dar. Mit diesen Daten dürfen alle Operationen durchgeführt werden, die für ihre Typen erlaubt sind, z. B.:

```
2*PI^ + 1
PA^[2] + 0.5
PR^.FA + 1.0
```

Der Deklarationszwang von PASCAL erfordert es, Bezeichner vor ihrer ersten Verwendung zu deklarieren. Beim Pointertyp gibt es die einzige Ausnahme. Sie ist notwendig, damit gleiche Datenobjekte miteinander „verzeigert" werden können.
Das Beispiel

```
TYPE Zeiger = ^ Knoten;
     Knoten = RECORD
       Info:...;
       Nachfolger: Zeiger
     END;
```

kann ohne Vorgriff nicht gelöst werden
Falls der Datentyp Zeiger nicht noch in anderen Deklarationen benötigt wird, kann auch geschrieben werden:

```
TYPE Knoten = RECORD
       Info:...;
       Nachfolger: ^ Knoten
     END;
```

Auch hier wird bereits auf Knoten Bezug genommen, obwohl seine Deklaration noch nicht beendet ist.

### 10.3. Arbeit mit Pointern

Für Pointervariablen gleichen Typs ist die Zuweisung und die Ausführung der Vergleichsoperationen = und <> erlaubt. Mit dem Datenobjekt, auf das ein Pointer verweist, können die für seinen Datentyp zulässigen Operationen durchgeführt werden.
Es gibt die vordeklarierte Konstante NIL, die mit allen Pointertypen verträglich ist. Falls ein Pointer den Wert NIL hat, so zeigt er auf kein Datenelement. Der Zugriff über eine Pointervariable, die den Wert NIL hat, ist ein Fehler und führt zum Programmabbruch.
Für die Arbeit mit Pointer ist die Standardprozedur

```
NEW(p)
```

notwendig. Ihr aktueller Partner ist eine Pointervariable beliebigen Typs. Durch NEW wird für ein durch die Pointervariable referenziertes Datenobjekt Speicher bereitgestellt. Seine Adresse wird zum Wert der Pointervariablen. Den bereitzustellenden Spei-

cher entnimmt NEW dem sogenannten Heap.
Das folgende Programmstück zeigt ein Beispiel:

```
TYPE T = ARRAY[1...10] OF REAL;
VAR P:^T;
BEGIN
  ...
  NEW(p);
  P^[1]:=1.5;
  ...
```

Vor dem Aufruf von NEW darf über P^ nicht zugegriffen werden. Erst durch NEW wird ein Array mit 10 REAL-Zahlen im Heap angelegt und kann anschließend genutzt werden.
Ob eine Pointervariable am Programmstart mit dem Wert NIL initialisiert ist, hängt vom verwendeten PASCAL-System ab. In TURBO-PASCAL ist ihr Wert unbestimmt. Durch die ersten Anweisungen eines Programms sollte Pointervariablen deshalb generell NIL zugewiesen werden.
Die nachfolgende Prozedur zeigt das Verzeigern von Datenobjekten zu einer Liste.

```
TYPE Zeiger = ^Knoten;
     Knoten = RECORD
      Info: Infotyp;
      Nachf: Zeiger
     END;

PROCEDURE Einf(V: Infotyp;
VAR P: Zeiger;
BEGIN
 NEW(p);
 WITH P^ DO BEGIN
  Info: = V;
  Nachf: = Anfang
 END;
 Anfang: = P
END;
```

Die einzelnen Elemente der Liste sind vom Typ des schon bekannten Records Knoten. Das Feld Info enthält die Daten. Nachf zeigt auf das folgende Listenelement. Das Einfügen eines Elements geschieht am Listenanfang und wird durch die Prozedur Einf ausgeführt. Sie erhält dazu als Parameter die einzufügende Information und einen Zeiger auf den Listenanfang. Da der Listenanfang modifiziert wird, muß er als Referenzparameter übergeben werden. Zuerst wird ein neues Listenelement angelegt. Danach wird die Information eingetragen und die bisherige Liste als Nachfolger angehängt. Anschließend wird der Anfang auf das eben angelegte Element gestellt.
Bei Variantenrecords ist es möglich, aus dem Heap nur soviel Speicher auszufassen, wie tatsächlich benötigt wird. Dazu erhält NEW weitere Parameter:

**NEW (p, t1, t2, ...);**

Die ti repräsentieren die Werte der Anzeigefelder der anzulegenden Variante.
Dazu ein Beispiel:

```
VAR p:^RECORD
   CASE BOOLEAN OF
   FALSE: (C:CHAR);
   TRUE: (R:ARRAY[1...10] OF REAL)
  END;
```

Beim Aufruf von

**NEW (p, FALSE)**

wird nur ein Datenelement vom Typ CHAR im Heap angelegt, bei

**NEW (p, TRUE)**

ein Array von 10 REAL-Zahlen.
In TURBO-PASCAL können bei NEW keine Werte für Anzeigefelder angegeben werden. NEW faßt immer den maximalen Speicherplatz aus.

### 10.4. Programmbeispiel

Das Programm löst die Aufgabe, Namen entsprechend ihrer lexikographischen Ordnung zu sortieren. Dazu wird mit den Namen eine Baumstruktur aufgebaut. Folgender Record bildet einen Knoten dieses Baumes:

```
TYPE Knoten = RECORD
     Name: Alpha;
     Vorg: ^Knoten;
     Nachf: ^Knoten
    END;
```

Das erste Feld trägt den Namen. Die Felder Vorg und Nachf verweisen auf weitere Knoten, die so in den Baum eingeordnet sind, daß

**Vorg^.Name < Name < Nachf^.Name**

gilt. Falls es keine Vorgänger und/oder Nachfolger gibt, hat das entsprechende Feld den Wert NIL. Das entsprechende Programm wird im Bild 10.2 gezeigt. Zu Beginn der Arbeit ist der Baum leer, das heißt, der auf ihn verweisende Zeiger hat den Wert NIL. Durch die Prozedur Insert wird der Baum Knoten für Knoten aufgebaut. Sie steigt zunächst solange im Baum ab; bis eine leere Stelle gefunden wird, an die der einzuordnende Name entsprechend seiner lexikographischen Ordnung paßt. Dort fügt sie einen neuen Knoten ein.

```
TYPE
  alpha=ARRAY[1..10] OF CHAR;
  link=^node;
  node=RECORD
    name: alpha;
    Vorg: link;
    Nachf:link
  END;


PROCEDURE Insert(VAR Teilbaum:link; Info:alpha);
BEGIN
  IF Teilbaum=NIL THEN BEGIN
    NEW(Teilbaum);
    WITH Teilbaum^ DO BEGIN
      name:=Info; Vorg:=NIL; Nachf:=NIL
    END
  END ELSE IF Info<Teilbaum^.name THEN
    Insert(Teilbaum^.Vorg,Info)
  ELSE
    Insert(Teilbaum^.Nachf,Info)
END;

PROCEDURE Print(Teilbaum:link);
BEGIN
  IF Teilbaum<>NIL THEN BEGIN
    Print(Teilbaum^.Vorg);
    WRITELN(Teilbaum^.name);
    Print(Teilbaum^.Nachf)
  END
END;
```

*Bild 10.2 Prozeduren zur Arbeit mit einem binären Baum*

Die Prozedur Print traversiert den Baum. Sie druckt für jeden Knoten zuerst die Namensfelder der Vorgängerknoten, dann den Namen des Parameterknotens und danach die Namen der Nachfolgeknoten.
Die Prozeduren Insert und Print rufen sich rekursiv auf. In TURBO-PASCAL müssen sie deshalb mit der Compileroption ◌A- übersetzt werden.

*10.5. Verwaltung des Heap-Speichers*

Durch weitere Standardprozeduren ist es möglich, nicht mehr benötigten Speicherplatz wieder in den Heap einzugliedern. Er kann so im Zuge der Abarbeitung des Programms wieder für andere Datenobjekte benutzt werden.
Zur effektiven Anwendung dieser Prozeduren ist es erforderlich, die Anordnung eines PASCAL-Programms im Hauptspeicher zu kennen. Sie wird im Bild 10.3 gezeigt. Aus prinzipiellen Gründen gibt es zwischen den PASCAL-Systemen kaum Unterschiede. Der Speicherbereich untergliedert sich zunächst in Kode und Daten. Der Kodebereich enthält das Hauptprogramm und die Prozeduren.
Der Datenbereich untergliedert sich weiter in Heap. Stack und Hauptprogrammvariable.

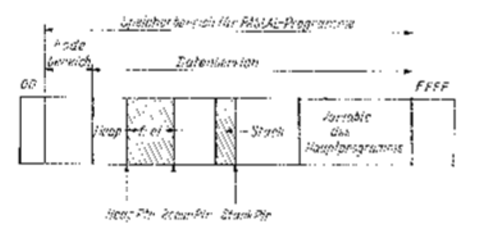



***Bild 10.3 Prinzipielle Anordnung eines PASCAL-Programms im Hauptspeicher***

Im Stack werden Parameter für Prozeduren und Funktionen sowie Zwischenresultate der Ausdrucksberechnung abgelegt.
Der Heap wächst nach steigenden, der Stack nach fallenden Adressen. Die PASCAL-Systeme überwachen die Ausdehnung von Stack und Heap. Ein Ineinanderlaufen von Stack und Heap führt zum Programmabbruch mit einer entsprechenden Fehlermeldung.
Bei TURBO-PASCAL gibt es noch einen Rekursionsstack, der mit Hilfe von RecurPtr verwaltet wird. In ihm sichern die rekursiv aufrufbaren Prozeduren die „alten" Werte ihrer lokalen Variablen. In TURBO-PASCAL sind HeapPtr, RecurPtr und StackPtr vordeklarierte Variablen, denen das PASCAL-Programm Werte zuweisen kann.
Ein fehlerfreies Arbeiten ist aber nur dann gewährleistet, wenn stets gilt:

**HeapPtr** < RecurPtr < StackPtr

Die Standardprozedur

**DISPOSE(p)**

gibt den Speicherplatz wieder frei, auf den die Pointervariable p verweist. Das eigentliche Problem bei DISPOSE liegt in der sinnvollen Wiederverwendung von freien Speicherbereichen, die sich inmitten des Heap befinden.

Beim TURBO-PASCAL wird darüber Buch geführt. Die Standardprozedur NEW versucht zuerst, das neu anzulegende Datenobjekt in einer Lücke zu plazieren. Die Standardfunktionen MEMAVAIL und MAXAVAIL liefern die Menge des insgesamt bzw. zusammenhängend feien Speichers in Byte.
Die Standardprozeduren

**MARK (p) und**
**RELEASE (p)**

eröffnen dem Programmierer den Zugang zu einer eigenen Heapverwaltung. MARK (p) legt den augenblicklichen Wert von HeapPtr in der Pointervariablen p ab. Durch RELEASE (p) wird HeapPtr wieder auf den Wert der Pointervariablen p gesetzt.
Es unterliegt der Verantwortung des Programmierers, zu entscheiden, wann HeapPtr zurückgesetzt wird.
Die Prozeduren MARK und RELEASE stellen die einfachste Form der Heap-Verwaltung dar. Sie gibt es in nahezu allen PASCAL-Systemen.
TURBO-PASCAL hat weiterhin die Prozeduren

**GETMEM (p, n) und**
**FREEMEM (p, n).**

Sie fassen aus dem Heap n Bytes aus bzw. geben ihn zurück. Mit GETMEM läßt sich der Mangel von NEW für Variantenrecords mindern.

## 11. TURBO-PASCAL-Sytem

### 11.1. Einführung

Gegenstand dieses Kapitels ist die Darstellung von *Erweiterungen und Leistungen* des TURBO-PASCAL-Systems, die nicht Bestandteil von Standard-PASCAL sind und in dieser Form nicht oder nur teilweise in anderen PASCAL-Systemen anzutreffen sind. Das sind:
- Typenübertragung
- typisierte Konstanten
- Datentypen BYTE und STRING
- absolute Variablen
- vordeklarierte Arrays für Zugriff auf den Hauptspeicher und die E/A-Ports
- Einfügen von Maschinenkodepassagen
- Aufruf von Maschinenkodeunterprogrammen
- Überlagerungsstrukturen für PASCAL-Prozeduren
- unterschiedliche Modi der Objektkodeerzeugung
- Quellprogrammuntergliederung.

Weiterhin wird im letzten Abschnitt die Bedienung des Systems erläutert.

### 11.2. Typenübertragung

Die Möglichkeit der Typenübertragung gestattet es dem Programmierer, gezielt die *Typenkontrolle des Compilers außer Kraft* zu setzen und in bestimmtem Umfang normalerweise unverträgliche Datenobjekte einander zuzuweisen.
Die Anwendung dieses Mechanismus wird an zwei Beispielen gezeigt. Im ersten Beispiel wird ein Datenelement vom Typ BOOLEAN einem INTEGER-Datenelement zugewiesen:

```
VAR i: INTEGER;
    b: BOOLEAN;
...
i:= INTEGER(b);
```

Das zweite Beispiel zeigt die Anwendung bei Aufzählungstypen:

```
TYPE farbe = (rot, blau, grün);
VAR f: farbe; i: INTEGER;
...
f: = farbe(0);
```

Entsprechend der internen Darstellung von Aufzählungstypen ist die letzte Zuweisung mit

```
f: = rot
```

äquivalent.
Bei der Typenübertragung wird nicht konvertiert. Das heißt, die Typenübertragung z. B. zwischen REAL und INTEGER ist nicht möglich.

### 11.3. Typisierte Konstanten

Ein oft diskutierter Mangel an Standard-PASCAL ist das Fehlen von Ausdrucksmitteln, mit denen *strukturierte Konstanten* deklariert werden können.
TURBO-PASCAL stellt zur Lösung dieses Problems sogenannte typisierte Konstanten bereit.
Mit Hilfe der nach Bild 11.1 modifizierten Konstantendeklaration werden sie in die Sprache eingebaut. Die Bilder 11.2 bis 11.5 zeigen weitere Einzelheiten der Syntax.
Eine naheliegende Anwendung könnte folgende Konstantendeklaration sein:

```
TYPE Tag = (Mo, Di, Mi,
            Do, Fr, Sa, So);
CONST WoAnfang: Tag = Mo;
```

Weitaus wichtiger dürfte aber die Deklaration von Array-, Record- und Mengenkonstanten sein. Es folgen Beispiele:




*Bild 11.1 Modifiziertes Syntaxdiagramm „konstantendeklaration*

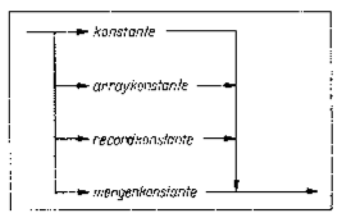



*Bild 11.2 Syntaxdiagramm „strukturierte konstante“*

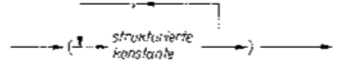



*Bild 11.3 Syntaxdiagramm „arraykonstante“*

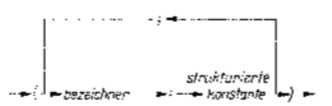



*Bild 11.4 Syntaxdiagramm „recordkonstante“*

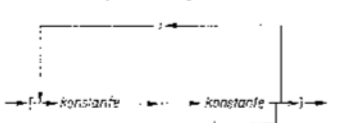



*Bild 11.5 Syntaxdiagramm „mengenkonstante“*

**a) Deklaration von Arraykonstanten**

```
TYPE Vekt = ARRAY [1...2] OF REAL;
     Mat = ARRAY [1...3, 1..2]
                         OF REAL;
CONST X: Vekt = (1.0, 2.0);
      A: Mat = ((1.1, 1.2),
                (2.1, 2.2),
                (3.1, 3.2));
```

**Die Werte der Arraykomponenten müssen zeilenweise aufgeführt werden.**

**b) Deklaration von Recordkonstanten**

```
TYPE REC = RECORD
     F1: INTEGER;
     F2: CHAR
    END;
CONST R: REC = (F1:1; F2:'A');
```

Die Recordfelder werden durch Voranstellen ihrer Bezeichner gekennzeichnet.

**c) Mengenkonstanten**

```
TYPE S = SET OF (A, B, C);
CONST S1: S = [A, C];
      S2: S = [A];
```

Die nun folgenden Beispiele zeigen die Kombination der Grundelemente:

```
TYPE T1 = ARRAY [1..2] OF
          RECORD
           F1: INTEGER;
           F2: CHAR
          END;
CONST K1: T1 = ((F1:1);
                (F2:'A');
                (F1:2);
                (F2:'B'));

TYPE T2 = RECORD
     F1: ARRAY [1..2]
           OF INTEGER;
     F2: ARRAY [1..3]
              OF CHAR
    END;
CONST K2: T2 = (F1: (1, 2);
                F2: ('A', 'B', 'C'));

TYPE T3 = ARRAY [1..2]
            OF SET OF 0..7;
CONST K3: T3 = ([0, 1], [6, 7]);
```

Für Arrays mit dem Komponententyp CHAR gibt es zwei mögliche Formen:

```
TYPE T = ARRAY [1 .. 3] OF CHAR;
CONST K1 : T = ('A', 'B', 'C');
      K2 : T = ('ABC');
```

Wichtig bei der Verwendung typisierter Konstanten ist, daß allen Komponenten eines Arrays oder allen Feldern eines Records Anfangswerte zugewiesen werden müssen.
Typisierte Konstanten dürfen wie Variablen benutzt werden, das heißt, abweichend von der Konzeption einer Konstante darf ihnen während der Programmabarbeitung ein Wert zugewiesen werden.

### 11.4. Datentypen BYTE und STRING

TURBO-PASCAL stellt zusätzlich die Datentypen BYTE und STRING bereit.
BYTE ist der *Teilbereich* 0..255 von INTEGER.
STRING ist ein Datentyp, dessen Wertebereich *Zeichenketten* einer bestimmten Länge sind, die bei der Deklaration angegeben werden muß. Die maximale Länge ist 255.
Variablen vom Stringtyp dürfen einander zugewiesen und verkettet werden. Als Verkettungsoperator fungiert das Pluszeichen.
Weiterhin ist die Anwendung der Vergleichsoperatoren zulässig.
Beispiele für Operationen mit Zeichenketten sind:

```
VAR S1, S2 : STRING [20];
...
S1 := 'ABC'
S2 := S1 + '.TXT';
IF S2 <> 'STOP' THEN ...
IF S1 < S2 THEN ...
```

Ein einzelnes Zeichen einer Zeichenkette kann durch Angabe seines Index ausgewählt werden. Die Zählung beginnt bei 1. Falls S1 z. B. den Wert 'ABC' hat, liefert S1 [3] das Zeichen 'C'.
Weiterhin dürfen Variablen vom Typ STRING als aktuelle Parameter der Standardprozeduren READ, WRITE und ASSIGN angegeben werden.
Das nachfolgende Beispiel zeigt dies:

```
VAR FileName: STRING [10];
    F: FILE OF ...;
.
.
READ (FileName);
ASSIGN (F, FileName);
.
.
```

Mit READ (FileName) wird der Name des durch ASSIGN zuzuweisenden Files erst zur Programmlaufzeit eingelesen.
Bei der internen Darstellung des Stringtyps wird die maximale Länge mitgeführt. Der Stringtyp STRING [Max] entspricht intern folgendem Rekord:

```
RECORD
  Laenge: BYTE;
  Info: ARRAY [1.. Max] OF CHAR
END
```

Falls die aktuelle Länge kleiner als Max ist, wird die Information durch eine Null beendet.
Zur Arbeit mit Stringvariablen gibt es Standardprozeduren und -funktionen. In der nachfolgenden Aufzählung sind s, s1, s2, ... Stringvariablen.

| | |
|---|---|
| **LENGTH (s)** | Liefert die aktuelle Länge von s. |
| **POS (s1, s2)** | Sucht s1 und s2 und liefert die Position. |
| **COPY (s, pos, n)** | Liefert Stringvariable, die von pos beginnend aus n Zeichen von s gebildet wird. |
| **CONCAT (s1, s2, ...)** | Liefert Stringvariable aus Kettung von s1, s2, .. |
| **DELETE (s, pos, n)** | Entfernt aus s ab pos n Zeichen. |
| **INSERT (s1, s2, pos)** | Fügt s1 ab pos in s2 ein. |
| **STR (ausdruck, s)** | Konvertiert ausdruck in Stringvariable s. |
| **VAL (s, variable, pos)** | Konvertiert s in die interne Darstellung von variable. Enthält s ein nicht erlaubtes Zeichen, zeigt pos seine Position an. |

### 11.5. Absolute Variablen

Mit absoluten Variablen kann die Speicherplatzordnungsstrategie des Compilers umgangen und einer Variablen eine *vorgegebene Speicheradresse* zugeordnet werden.
Die folgenden Beispiele zeigen die Deklaration des Bildwiederholspeichers als zweidimensionales Array von CHAR und den Zugriff auf die Kommandozeile des Betriebssystems:

```
VAR BWS : ARRAY [1 .. 24, 1 .. 80]
        OF CHAR ABSOLUTE ○F800;
    .
    .
    KomZeile: STRING [127]
              ABSOLUTE ○80;
    .
    .
WRITELN ('KomZeile: ',Cmd)
```

(Der Zugriff auf die Kommandozeile funktioniert nur, falls das PASCAL-Programm als COM-File durch das Betriebssystem gestartet wurde).
Eine weitere Möglichkeit, in die Speicherplatzzuordnungsstrategie des Compilers einzugreifen, ist das „Übereinanderlegen" von Variablen.

```
VAR XREAL : REAL
    XArr : ARRAY [1 .. 6] OF BYTE
           ABSOLUTE XReal;
```

Die Variablen XReal und XArr haben die gleiche Position im Hauptspeicher.

### 11.6. Vordeklarierte Arrays

TURBO-PASCAL stellt die vordeklarierten Arrays

```
VAR MEM : ARRAY [0 .. ○FFFF]
              OF BYTE;
    PORT : ARRAY [0 .. ○FF]
              OF BYTE;
```

bereit. MEM korrespondiert mit dem *Hauptspeicher* und PORT mit den *E/A-Toren* des Rechners.
Die Anweisungen
PORT [n]: = ...;
... : = PORT [n];
werden in die Maschinenbefehle
OUT n und
IN n
umgesetzt.

### 11.7. Einfügen von Maschinenkodepassagen

Maschinenkodepassagen können mit der Inline-Anweisung wie eine gewöhnliche Anweisung in ein PASCAL-Programm eingefügt werden. Die Maschinenbefehle sind in ihrer numerischen Darstellung anzugeben.
Zur Erleichterung dürfen jedoch Variablenbezeichner verwendet werden. Der Compiler ersetzt sie durch ihre Adressen. In TURBO-PASCAL ist das problemlos möglich, da allen Variablen eine feste Hauptspeicheradresse zugeordnet ist (siehe Pkt. 5.5.4.). Eine Bezugnahme auf den Speicherplatzzuweisungszähler ist mit Hilfe des Zeichens "*" möglich.
Das folgende Programmstück zeigt den Befehl LD HL, (X):

```
VAR X: INTEGER
...
INLINE (○2A/X);
...
```

Die einzelnen Bytes bzw. Variablenbezeichner sind durch Schrägstriche zu trennen.

### 11.8. Aufruf von Maschinenkodeunterprogrammen

TURBO-PASCAL gestattet den Aufruf von Maschinenkodeunterprogrammen. Dazu ist wie bei einer PASCAL-Prozedur die Angabe eines Prozedurkopfes notwendig. Der Block wird jedoch durch den reservierten Bezeichner EXTERNAL und die Adresse des Maschinenkodeprogramms ersetzt.
Nachfolgend ein Beispiel für eine Prozedur mit zwei Parametern:

```
PROCEDURE MaschKode
    (VAR A: INTEGER; B: REAL);
        EXTERNAL adresse;
```

Aufgerufen wird das Maschinenkodeunterprogramm wie eine PASCAL-Prozedur.
Vom Programmierer ist abzusichern, daß zur Abarbeitungszeit an der angegebenen Speicheradresse tatsächlich das erwartete Programm steht.

*wird fortgesetzt*

# PASCAL (Teil 6)

**Dr. Claus Kofer**
**Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden**

In dem Maschinenkodeprogramm sind Parameterübergabe- und Aufrufmechanismus des TURBO-PASCAL-Systems nachzubilden. Die Parameterübergabe geschieht über den Stack. Die Parameter werden, beim ersten beginnend, durch PUSH im Stack abgelegt. Der Aufruf erfolgt durch CALL. Vor Verlassen des aufgerufenen Unterprogramms müssen alle Parameter vom Stack entfernt werden. Dabei darf natürlich die zuoberst stehende Rücksprungadresse nicht verlorengehen.
Referenzparameter nehmen im Stack zwei Byte ein. Sie sind die Adresse der entsprechenden Variablen. Records und Arrays werden ebenfalls stets wie Referenzparameter übermittelt. Werteparameter benötigen Platz entsprechend ihres Datentyps.

### 11.9. Überlagerungsstrukturen

Das TURBO-PASCAL-System bietet die Möglichkeit, *einzelne Prozeduren* des PASCAL-Programms in einer *Überlagerungsstruktur* anzuordnen. Eine solche Prozedur wird durch Voranstellen des reservierten Bezeichners OVERLAY gekennzeichnet.
Prozeduren, die sich überlagern sollen, müssen im Deklarationsteil aufeinanderfolgen. Im Kodebereich des PASCAL-Programms wird für sie nur soviel Speicher vorgesehen, wie für die Aufnahme der größten von ihnen erforderlich ist. Nachfolgend ein Beispiel:

```
PROGRAM Ovl;
PROCEDURE P1;
.
.
.
END;
OVERLAY PROCEDURE P2;
.
.
.
END;
OVERLAY PROCEDURE P3;
.
.
.
END;
PROCEDURE P4;
.
.
.
END;
BEGIN
.
.
.
END.
```

Das Bild 11.6a zeigt die Anordnung der Prozeduren und des Hauptprogramms im Kodebereich. Die Überlagerungsprozeduren P2 und P3 nehmen denselben Speicherbereich ein. Entgegen den Gültigkeitsregeln für Bezeichner kann P2 jedoch nicht durch P3 aufgerufen werden.

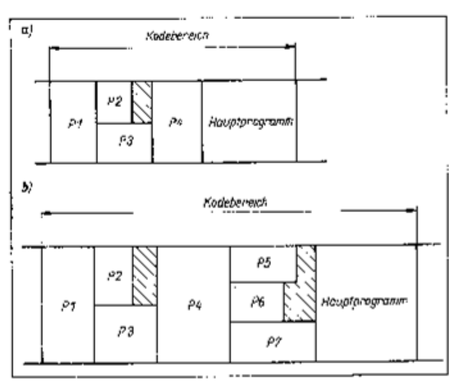



***Bild 11.6 Anordnung von Überlagerungsprozeduren im Kodebereich***

Auch die Bildung mehrerer Gruppen von Überlagerungsprozeduren ist möglich. Zur Demonstration wird das obige Programm erweitert:

```
...
PROCEDURE P4;
.
.
.
END;
.
.
OVERLAY PROCEDURE P5;
.
.
.
END;
OVERLAY PROCEDURE P6;
.
.
.
END;
OVERLAY PROCEDURE P7;
.
.
.
END;
BEGIN
.
.
.
END.
```

Das Aussehen des Kodebereiches zeigt Bild 11.6b.

Die einzelnen Gruppen von Überlagerungsprozeduren werden als separate Files auf der Diskette abgelegt. Sie bekommen den Namen des entsprechenden COM-Files, aber die Erweiterungen .000, .001 usw.
Das Laden der aufgerufenen Überlagerungsprozeduren organisiert das PASCAL-System. Dabei wird vor dem Diskettenzugriff überprüft, ob sich die gerufene Prozedur bereits im Speicher befindet.

### 11.10. Modi der Objektkodeerzeugung des Compilers

Beim TURBO-PASCAL-Compiler können drei verschiedene Modi der Objektkodeerzeugung eingestellt werden:
– Hauptspeichermodus
– COM-Filemodus
– Kettenfilemodus.

Im Hauptspeichermodus wird der Objektkode im Hauptspeicher abgelegt. Dieser Modus ist für die Programmentwicklung vorgesehen. Für Produktionsanwendungen ist er nicht geeignet, da zur Abarbeitung des PASCAL-Programms das gesamte TURBO-PASCAL-System erforderlich ist.
Im COM-Filemodus werden COM-Files des Betriebssystems erzeugt. Sie bestehen aus PASCAL-Laufzeitsystem, Programmkode und Programmvariablen. Das COM-File hält die Konventionen des Betriebssystems ein und unterscheidet sich nicht von anderen abarbeitbaren Programmen.
Im Kettenfilemodus werden sogenannte Kettenfiles erzeugt. Sie sind zur unmittelbar aufeinanderfolgenden Ausführung von PASCAL-Programmen vorgesehen. Kettenfiles sind den COM-Files sehr ähnlich. Bei ihnen fehlt lediglich das PASCAL-Laufzeitsystem. Die Abarbeitung eines Kettenfiles kann deshalb nur durch ein anderes PASCAL-Programm eingeleitet werden. Dafür gibt es die Standardprozedur
CHAIN(f).
f muß eine Filevariable sein, der vorher mit ASSIGN das zu startende Kettenfile zugewiesen wurde. Das Kettenfile verdrängt den Kode des rufenden Programms. Mit Hilfe von absoluten Variablen kann es mit Daten versorgt werden.
In gewissem Grade kann der Programmierer darauf Einfluß nehmen, wie sein PASCAL-Programm im Hauptspeicher plaziert wird. Das Bild 11.7 zeigt für die besonders wichtigen COM-Files die hier interessierenden Details. Das gesamte Programm beginnt stets am Anfang des TPA bei der Adresse 100H mit dem etwa 8 KByte großen Laufzeitsystem. An das Laufzeitsystem schließt sich der Kode der Prozeduren und Funktionen sowie des Hauptprogramms an. Standardmäßig beginnt der Kode auf dem ersten freien Byte hinter dem Laufzeitsystem. Es ist aber möglich, die Startadresse zu erhöhen, um

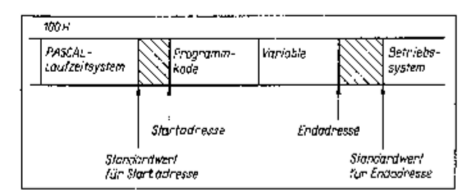



*Bild 11.7 Wirkung von Start- und Endadresse auf die Plazierung von PASCAL-Programmen im Hauptspeicher*

hier Platz freizulassen für absolute Variablen. Endadresse stellt die höchste für das PASCAL-Programm zu verwendende Adresse dar. Standardmäßig ist sie etwa 700 Byte kleiner als die obere Grenze des TPA. Die Endadresse kann verringert werden, um z. B. Programme zu erzeugen, die auf einem Rechner mit kleinerem TPA laufen können, oder um Platz für Maschinenkodeunterprogramme freizuhalten.

### 11.11. Quellprogrammuntergliederung

TURBO-PASCAL gestattet mit Hilfe der INCLUDE-Option des Compilers die *Untergliederung* eines PASCAL-*Quellprogramms* in mehrere getrennte Files. Durch sie wird sofort die Quelltexteingabe des Compilers auf das angegebene File umgelegt und dieses bis zum Ende gelesen. Das Bild 11.8 zeigt ein Beispiel.

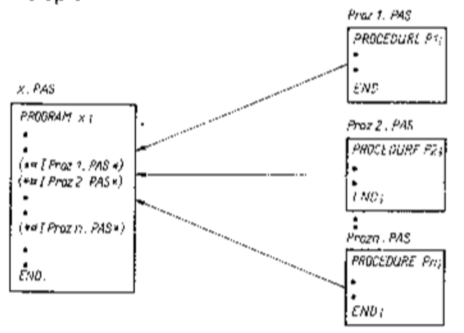



*Bild 11.8 Untergliederung eines PASCAL-Programms in mehrere Quelltextfiles*

TURBO-PASCAL unterscheidet zwischen *Haupt- und Arbeitsfile*. Der Compiler beginnt die Übersetzung stets mit dem Hauptfile.
Das System hält das Arbeitsfile in einem internen Quelltextbereich, in dem es mit dem Editor korrigiert werden kann.
Haupt- und Arbeitsfile werden zu Beginn der Arbeit vom Nutzer festgelegt. Während das Hauptfile nur durch den Nutzer verändert werden kann, wählt das TURBO-PASCAL-System das File zum neuen Arbeitsfile, bei dem während der Übersetzung ein Fehler auftrat.

### 11.12. Bedienung des TURBO-PASCAL-Systems

Die Bedienung des TURBO-PASCAL-Systems erfolgt mit Hilfe von einfachen prägnanten Menüs. Kommandoeingaben bestehen aus einem einzigen Buchstaben bzw. aus Filebezeichnungen und Adressen.
Die erlaubten Kommandos werden am Zustandsdiagramm des TURBO-PASCAL-Systems erläutert. Es wird im Bild 11.9 gezeigt. Nach dem Start geht das System in einen Anfangszustand. Hier kann über das Laden der Fehlertexte des Compilers entschieden werden. Sie beanspruchen einen Platz von etwa 1600 Byte.

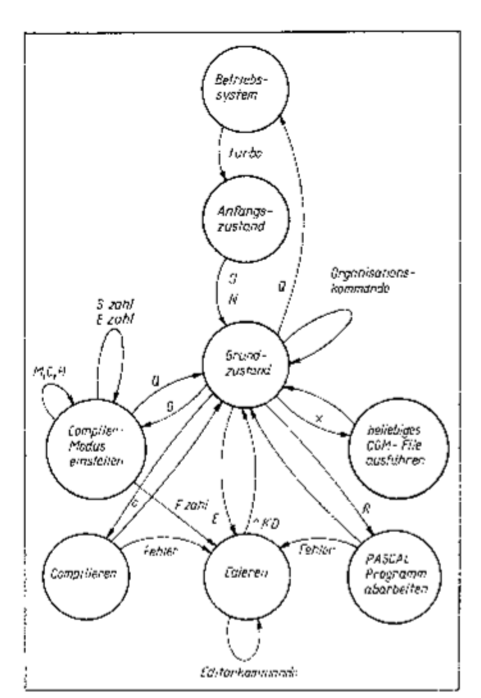



*Bild 11.9 Zustandsdiagramm und Bedienerkommandos des TURBO-PASCAL-Systems*

Im Grundzustand können zunächst folgende Organisationskommandos eingegeben werden:

| | |
|---|---|
| L | Wahl des aktuellen Laufwerkes (A:, B:, ...) |
| D | Anzeige Inhaltsverzeichnisinformationen |
| M | Angabe des Hauptfiles. Es wird zur Eingabe eines Filenamens aufgefordert. |
| W | Angabe des Arbeitsfiles. Es wird zur Eingabe eines Filenamens aufgefordert. |
| S | Abspeichern des Arbeitsfiles auf Diskette. |

Durch folgende Kommandos erfolgt der Übergang in die anderen Zustände:

| | |
|---|---|
| E | Edieren. Im Editorzustand ist die Eingabe von Korrekturkommandos möglich. Es wird die im Anhang 1 angezeigte Teilmenge der TP-Kommandos angeboten. Der Zustand wird durch ^KD verlassen. |
| C | Compilieren. Es wird das Hauptfile übersetzt. Falls keines angegeben wurde, nimmt der Compiler das Arbeitsfile. Bei Auftreten eines Fehlers wird sofort in den Editorzustand gewechselt und der Cursor auf die Fehlerstelle positioniert. |
| R | Starten des übersetzten PASCAL-Programms. Falls noch keines übersetzt wurde, wird eine Compilierung eingeschoben. Tritt bei der Programmabarbeitung ein Fehler auf, wird eine entsprechende Meldung ausgegeben, nach Betätigung der ESC-Taste in den Editorzustand gewechselt und der Cursor auf die Fehlerstelle positioniert. |
| X | Ausführung eines beliebigen COM-Files. Das TURBO-PASCAL-System wird bis auf einen kleinen residenten Teil im Speicher überschrieben. Nach Beendigung der Abarbeitung des COM-Files regeneriert der residente Teil den alten Zustand des TURBO-PASCAL-Systems wieder. |
| O | Compilermodus einstellen. In diesem Zustand ist die Eingabe weiterer Kommandos möglich: |
| M,C,H | Einstellung des Hauptspeicher-, COM-File- oder Kettenfilemodus |
| Sn | Setzen der Startadresse auf n |
| En | Setzen der Endadresse auf n |
| Fn | Suchen der Quelltextpassage, zu der die Kodeposition n gehört, und Wechsel in den Editorzustand |
| Q | Rückkehr in den Grundzustand. |

Das Kommando F n dient dazu, Laufzeitfehler in COM-Files zu lokalisieren.

### 11.13. Separate Übersetzung

Ab Version 4.0 bietet TURBO-PASCAL die Möglichkeit der separaten Übersetzung von einzelnen Teilen eines PASCAL-Programms unter voller Beibehaltung der Kontrollen des Compilers. Damit wird die Bildung von Programmbibliotheken für oft benutzte Prozeduren und Funktionen ermöglicht sowie die kollektive Erarbeitung von großen Softwareprojekten unterstützt.
Die für die separate Übersetzung erforderlichen Ausdrucksmittel fügen sich organisch in die Sprache ein. Bereits existierende PASCAL-Programme sind ohne Änderungen auch in Version 4.0 lauffähig.
Die Syntaxdiagramme der Bilder 11.10 bis 11.15 zeigen die erforderlichen Spracherweiterungen. Es wird entsprechend Bild 11.10 zunächst eine Übersetzungseinheit eingeführt. Sie ist alternativ das schon bekannte Programm oder eine Unit.
Den Aufbau einer Unit zeigt das Syntaxdiagramm Bild 11.11. Auf den reservierten Bezeichner *UNIT* folgen Unit-Name sowie Interface- und Implementationsteil. Den Aufbau des Interfaceteiles zeigt Bild 11.12. Er wird durch den reservierten Bezeichner *INTERFACE* eingeleitet und besteht aus einer – möglicherweise auch leeren – Folge von Deklarationen von Marken, Konstanten, Typen, Variablen sowie Prozedur- und Funktionsköpfen. Alle im Interfaceteil deklarierten Objekte haben globale Bedeutung. Sie können in einem Programm oder in anderen Units benutzt werden.
Den Aufbau des Implementationsteils zeigt Bild 11.13. Er wird durch den reservierten Bezeichner *IMPLEMENTATION* eingeleitet. Darauf folgt ein Block, der entsprechend Bild 5.1 aus Deklarations- und Anweisungsteil besteht. Seine Deklarationen sind jedoch aufgrund der Gültigkeitsregeln von PASCAL nur innerhalb des Implementationsteils gültig.
Enthält der Interfaceteil Prozedur- und Funktionsköpfe, dann muß im Implementationsteil die vollständige Deklaration dieser Prozeduren und Funktionen folgen.
Die Unit *Counter* demonstriert dies:

```
UNIT Counter;
INTERFACE
   PROCEDURE IncCount(amount: INTEGER);
```

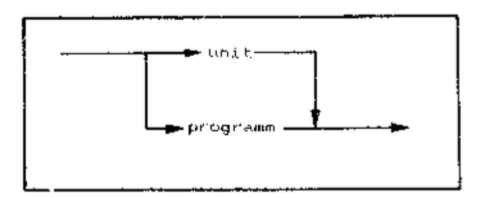



*Bild 11.10 Syntaxdiagramm „übersetzungseinheit“*

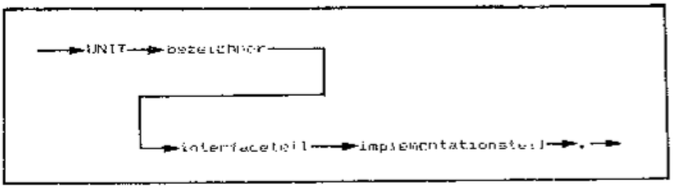



*Bild 11.11 Syntaxdiagramm „unit“*

*Bild 11.13 Syntaxdiagramm „implementationsteil“*

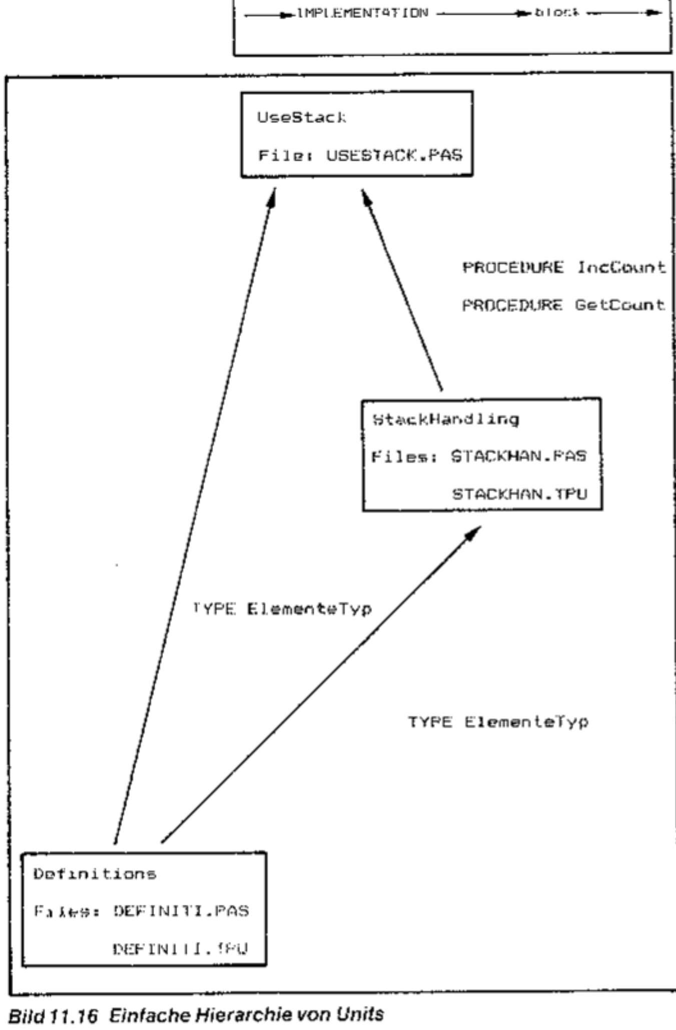



*Bild 11.16 Einfache Hierarchie von Units*

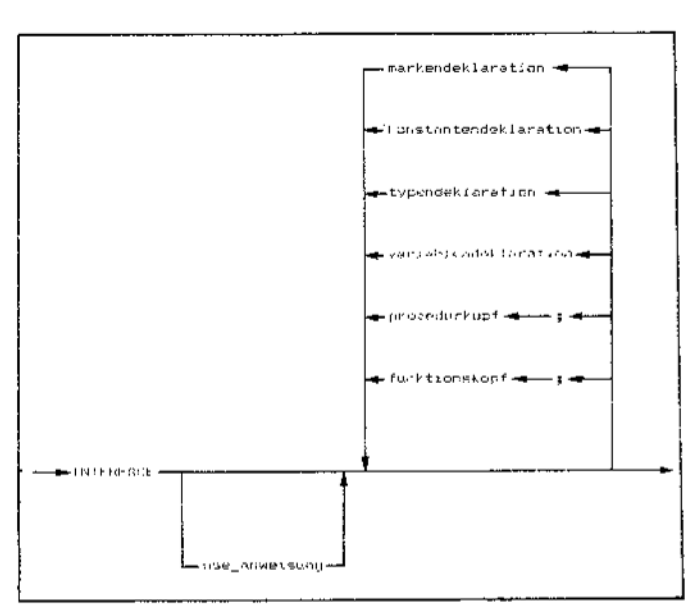



*Bild 11.12 Syntaxdiagramm „Interfaceteil“*

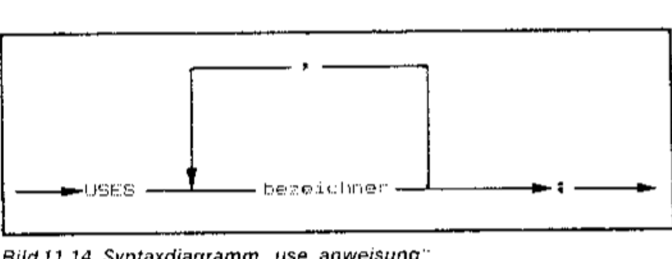
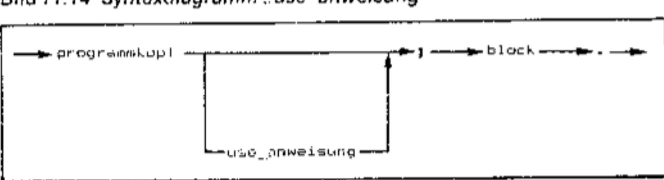



*Bild 11.14 Syntaxdiagramm „use anweisung“*




◄ *Bild 11.15 Modifiziertes Syntaxdiagramm „programm“*

```
   PROCEDURE GetCount(VAR i: INTEGER);
IMPLEMENTATION
   VAR Count: INTEGER;
   PROCEDURE IncCount(amount: INTEGER);
   BEGIN
      Count := Count + amount
   END;
   PROCEDURE GetCount(VAR i: INTEGER);
   BEGIN
      i := Count
   END;
   BEGIN
      Count := 0
   END.
```

Der Interfaceteil der Unit *Counter* wird durch die Köpfe der Prozeduren *IncCount* und *GetCount* gebildet. Dadurch können sie in anderen Übersetzungseinheiten benutzt werden. Der Implementationsteil enthält die Deklaration der Variablen *Count* und die vollständige Deklaration der Prozeduren *IncCount* und *GetCount*. Die Variable *Count* ist außerhalb des Implementationsteils und damit außerhalb der Unit *Counter* nicht bekannt. Der Anweisungsteil BEGIN Count:=0 END des Implementationsteils dient im Programmbeispiel zur Initialisierung der Zählvariablen *Count*. Das TURBO-PASCAL-System sichert ab, daß die Anwendungsteile aller Units vor dem Anweisungsteil des Programms, das die Units benutzt, abgearbeitet werden.
Eine Unit muß nicht notwendigerweise ausführbare Anweisungen enthalten. Im nachfolgenden Programmbeispiel wird durch die Unit *Definitions* lediglich der Datentyp *COMPLEX* deklariert:

```
UNIT Definitions;
INTERFACE
   TYPE
      COMPLEX = RECORD
         re: REAL;
         im: REAL
      END;
IMPLEMENTATION
BEGIN END.
```

Implementations- und Anweisungsteil dieser Unit werden nicht benötigt und sind deshalb leer.
Zur Benutzung der in Units deklarierten Objekte ist die im Bild 11.14 gezeigte Use-Anweisung erforderlich. Sie wird entsprechend Bild 11.15 in das bereits bekannte Syntaxdia-

gramm *Programm* eingefügt. Sie stellt eine Liste von Namen der verwendeten Units dar. Das folgende Programmbeispiel zeigt, wie die Objekte der Unit *Counter* benutzt werden können:

```
PROGRAM demo;
USES Counter;
VAR i: INTEGER;
BEGIN
  IncCount(3); (*...*) IncCount(4);
  (*...*)
  GetCount(i);Writeln('Zählwert: ',i);
END.
```

Im Programmbeispiel werden die Objekte der Unit *Counter* mit dem Namen referenziert, mit dem sie deklariert wurden. Im Beispiel die Prozeduren *IncCount* und *GetCount*. Es ist jedoch durchaus denkbar, daß das nutzende Programm selbst Objekte gleichen Namens deklariert. Nach den Gültigkeitsregeln von PASCAL überdecken diese dann die Bezeichner der benutzten Units. Würde das Programm Demo z. B. die Anweisungen
USES Counter;
VAR IncCount: BOOLEAN;
enthalten, dann überdeckt die Variable *IncCount* die Prozedur *IncCount* aus der Unit *Counter*. Werden in einem solchen Fall die Objekte der benutzten Units dennoch benötigt, so sind sie mit dem Namen der Unit zu qualifizieren. Das heißt, ihnen ist der durch einen Punkt getrennte Unit-Name voranzustellen. Aufrufe der Prozedur *IncCount* haben dann die Form Counter.IncCount(3).
Es sind einige Erläuterungen zur Implementation der separaten Übersetzung in TURBO-PASCAL zweckmäßig. Units werden durch den Compiler (im Disk-Mode) in ein File mit dem Typ TPU übersetzt und auf der Diskette abgelegt. Für die Übersetzung von Programmeinheiten, die diese Unit benutzen, wird das korrespondierende TPU-File benötigt. Sein Name wird durch den Compiler aus dem Unit-Namen und dem Typ TPU gebildet. Dieser einfache und klare Mechanismus führt zu der Einschränkung, daß der Filename mit dem Unit-Namen übereinstimmen muß. Für das vorangegangene Beispiel heißt dies, daß der Programmtext der Unit *Counter* im File *COUNTER.PAS* abgelegt werden muß. Der Compiler übersetzt es in das File *Counter.TPU*, welches zur Übersetzung von *DEMO.PAS* benötigt wird.
Das Bild 11.12 zeigt, daß im Interfaceteil einer Unit ebenfalls eine Use-Anweisung angegeben werden darf. Dadurch ermöglicht TURBO-PASCAL die hierarchische Strukturierung von Units.
Vom TURBO-PASCAL-System werden dem Anwender eine Reihe von fertigen Units zur Verfügung gestellt, die zum Beispiel zur Arbeit mit dem Bildschirm dienen oder den Aufruf von Betriebssystemleistungen ermöglichen.

## Anhang

### A. Editorkommandos

#### A.1 Cursor-Positionierung

| | |
|---|---|
| Zeichen links | ^S |
| Zeichen rechts | ^D |
| Wort links | ^A |
| Wort rechts | ^F |
| Zeile hoch | ^E |
| Zeile tief | ^X |
| Rollen hoch | ^Z |
| Rollen tief | ^W |
| Blättern hoch | ^R Bild zurück |
| Blättern tief | ^C Bild vorwärts |
| Zeilenanfang | ^Q^S |
| Zeilenende | ^Q^D |
| Bildanfang | ^Q^E |
| Bildende | ^Q^X |
| Fileanfang | ^Q^R |
| Fileende | ^Q^C |
| Blockbeginn | ^Q^B |
| Blockende | ^Q^K |
| Letzte Position | ^Q^P Rückkehr zur letzten Cursorposition, vor Suchen, Ersetzen oder Abspeichern |

#### A.2 Einfügen und Löschen

| | |
|---|---|
| Insert-Modus Ein/Aus | ^V |
| Löschen links | DEL |
| Löschen Zeichen | ^G |
| Löschen Wort rechts | ^T |
| Zeile einfügen | ^N |
| Zeile löschen | ^Y |
| Zeilenrest löschen | ^Q^Y löscht ab Cursor bis Zeilenende |

#### A.3 Block-Kommandos

| | |
|---|---|
| Blockbeginn | ^K^B |
| Blockende | ^K^K |
| Marken löschen | ^K^H |
| Wort markieren | ^K^T |
| Block kopieren | ^K^C Kopieren an Cursorposition |
| Block verschieben | ^K^V Verschieben an die Cursorposition |
| Block löschen | ^K^Y |
| Block lesen | ^K^R Text von einem angeforderten File lesen |
| Block schreiben | ^K^W Block in ein File schreiben |

#### A.4 Verschiedene Kommandos

| | |
|---|---|
| Edieren beenden | ^K^D Rückkehr zum TURBO-Grundmenü |
| Indent Ein/Aus | ^Q^I automatisches Einrücken der Zeilenanfänge |
| Restore Zeile | ^Q^L Alle Veränderungen in der Zeile werden rückgängig gemacht. |
| Suchen <Find> | ^Q^F Es wird zu suchendes Muster angefordert; Optionen: B rückwärts suchen G globales Suchen n Suchen des n-ten Auftretens U Ignorieren Groß-/Kleinbuchstaben W nur Worte suchen |
| Substituieren | ^Q^A Zu suchendes Muster und Text werden angefordert; Optionen: B rückwärts suchen G globales Suchen (top to down) n Suchen des n-ten Auftretens N Ersetzen ohne Fragen: (Replace (Y/N) ?) U Ignorieren Groß-/Kleinbuchstaben W nur Worte suchen Optionen ohne Zwischenraum schreiben und mit <CR> beenden. |
| Wiederholen | ^L Wiederholen des letzten ^Q^F- oder ^Q^A-Kommandos |
| Abort-Kommando | ^U sofortiges Abbrechen jedes Editorkommandos |

### B. Compiler-Options

| | |
|---|---|
| ○A−,○A+ | Retten und Rückspeichern der lokalen Variablen bei Prozeduraufruf ein- bzw. ausschalten, d. h., rekursive Prozeduraufrufe sind bzw. sind nicht möglich. |
| ○C+,○C− | Bei Tastatureingabe werden folgende CTRL-Zeichen interpretiert bzw. ignoriert: ^C Programmabbruch ^S Stop/Start der Bildschirmausgabe |
| ○I+,○I− | Ein-/Ausschalten der Reaktion des Laufzeitsystems auf E/A-Fehler |
| ○R−,○R− | Überprüfung der Einhaltung des Wertebereiches der Aufzählungs- und Teilbereichstypen ein- bzw. ausschalten (Solche Typen sind Arrayindizes.) |
| ○U+,○U− | Der Abbruch eines Programms durch Betätigen der CTRL-C-Taste wird ermöglicht bzw. verhindert. |
| ○V+,○V− | Bei Parametern vom Datentyp STRING muß die max. Länge des aktuellen mit der des formalen Parameters übereinstimmen bzw. darf unterschiedlich sein. |
| ○Wn | Festlegen der maximalen Verschachtelungstiefe von With-Anweisungen ($0 \leqq n \leqq 9$). In jeder Prozedur werden für With-Anweisungen 2*n Byte im Datenbereich reserviert. |
| ○X+,○X− | Der Zugriff auf Arraykomponenten wird bzw. wird nicht optimiert. |

Für alle Options gibt es einen Standard: ○A+, ○C+, ○I+, ○R−, ○U−, ○V+, ○W4, ○X−.

### C. Zusätzliche Standardprozeduren und -funktionen

#### C.1 Ein- und Ausgabe

| | |
|---|---|
| BLOCKREAD(f,buff,anz) | Lesen von anz Blöcken zu je 128 Byte der Variablen buff von File f. f muß ein typenloses File sein: VAR f: FILE; |
| BLOCKWRITE(f,buff,anz) | analog BLOCKREAD |
| CHAIN (f) | Verdrängen des rufenden PASCAL-Programms durch das auf dem Kettenfile f abgespeicherte PASCAL-Programm. f muß ein typenloses File und vorher mit ASSIGN zugewiesen worden sein. |
| EXECUTE(f) | Wie CHAIN(f), jedoch darf f ein beliebiges COM-File sein. |
| ERASE(f) | Löschen des f zugewiesenen Files |
| FLUSH(f) | Leeren des dem File zugeordneten Puffers. |
| RENAME(f,string) | Das f zugewiesene File erhält den neuen in string angegebenen Namen. |

#### C.2 Bildschirmausgabe und Tastatureingabe

| | |
|---|---|
| CLREOL | Löschen einer Bildschirmzeile ab Cursorposition |
| DELLINE | Löschen der gesamten Zeile, in der der Cursor steht |
| INIT | Ausgabe der Terminalinitialisierungsfolge |
| INSLINE | Einfügen einer Leerzeile an Cursorposition |
| KEYPRESSED | Liefert TRUE, falls ein Zeichen im Eingabepuffer der Tastatur |

*Schluß*